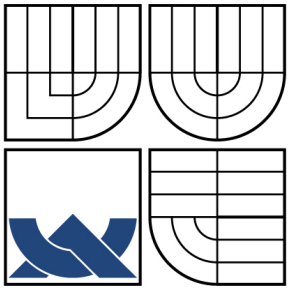
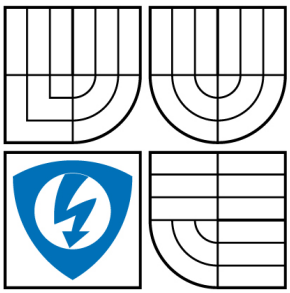


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF TELECOMMUNICATIONS

METODY PŘÍSTUPU K DATABÁZÍM POSTGRESQL V .NET FRAMEWORK

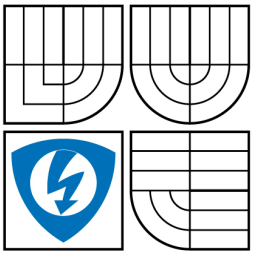
METHODS OF ACCESS TO POSTGRESQL DATABASES IN .NET FRAMEWORK

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. VÁCLAV HENZL
AUTHOR

VEDOUCÍ PRÁCE — doc. Ing. IVO LATTENBERG, Ph.D.
SUPERVISOR

BRNO 2009



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Diplomová práce

magisterský navazující studijní obor
Telekomunikační a informační technika

Student:	Bc. Václav Henzl	***ID:***	84379
Ročník:	2	***Akademický rok:***	2008/2009

NÁZEV TÉMATU:

Metody přístupu k databázím PostgreSQL v .NET Framework

POKYNY PRO VYPRACOVÁNÍ:

Vyberte nejvhodnější řešení přístupu k databázím PostgreSQL. Naprogramujte generátor kódu mapující jednotlivé tabulky v databázi na objekty v jazyce C#. Výsledkem generátoru by měl být generovaný zdrojový kód obsahující vygenerované objekty. Navrhněte dále obecný grid, který bude virtuálně zobrazovat obsah databázových tabulek a bude umožňovat práci s tabulkami s velkým množstvím záznamů. Vytvořte aplikaci, která bude demonstrovat použití vygenerovaných objektů a navrženého gridu.

DOPORUČENÁ LITERATURA:

[1] ECKEL, B., Myslíme v jazyku C++. Grada Publishing, Praha 2000. ISBN 80-247-9009-2. 552 stran.
[2] ŠIMŮNEK, M. SQL: kompletní kapesní průvodce. 1. vyd. Praha: Grada Publishing, s.r.o., 1999. 248 s. ISBN 80-7169-692-7.
[3] STANEK, W.R., Microsoft SQL Server 2005 Kapesní rádce administrátora. Computer press, ISBN: 0-7356-2107-1.

Termín zadání:	9.2.2009	***Termín odevzdání:***	26.5.2009

Vedoucí práce: doc. Ing. Ivo Lattenberg, Ph.D.

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

Abstrakt

Výsledkem této práce jsou dva hlavní projekty – NpgObjects a PagedDataGridView.

NpgObjects je jednoduchý ORM framework umožňující mapování databázových tabulek do objektů prostředí CLR platformy .NET. Pomocí speciálně navrženého generátoru jsou na základě informací o databázi vytvořeny třídy v jazyce C#. Tyto třídy mapují databázové tabulky jedna ku jedné. NpgObjects umožňuje pomocí vygenerovaných objektů všechny základní databázové operace – SELECT, INSERT, UPDATE a DELETE.

PagedDataGridView je komponenta pro zobrazování tabulkových dat. Ve spolupráci s NpgObjects umí stránkovat databázová data a tak řídit tok dat do aplikace. Poskytuje příjemné uživatelské rozhraní, pomocí kterého lze snadno navigovat mezi jednotlivými stránkami dat.

Klíčová slova

PostgreSQL, databáze, Npgsql, .NET Framework, C#, ORM, objekt, relace, mapovaní, DataGrid, tabulka, stránkování

Abstract

The results of this work are two major projects - NpgObjects and PagedDataGridView.

NpgObjects is a simple ORM framework to enable the mapping database tables to objects in the common language runtime. It contains a specially designed generator which generates classes in C# from information obtained from the database. These classes are mapping on the database tables one to one. NpgObjects allows all the basic database operations - SELECT, INSERT, UPDATE and DELETE.

PagedDataGridView is a component for displaying tabular data. In cooperation with NpgObjects can paginate database data and manage the flow of data into application. It provides a comfortable user interface, which can easily navigate between different pages of data.

Keywords

PostgreSQL, database, Npgsql, .NET Framework, C#, ORM, object, relation, mapping DataGrid, table, paging

HENZL, V. Metody přístupu k databázím PostgreSQL v .NET Framework. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2009. 101 s. Vedoucí diplomové práce doc. Ing. Ivo Lattenberg, Ph.D.

Prohlášení

Prohlašuji, že svou diplomovou práci na téma *Metody přístupu k databázím PostgreSQL v .NET Framework* jsem vypracoval samostatně pod vedením vedoucího semestrálního projektu a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne …… ……

podpis autora

Obsah

Seznam obrázků

Seznam tabulek

Úvod

PostgreSQL je výkonný open-source relační databázový systém s dlouhou tradicí, který nachází uplatnění v mnoha odvětvích. Díky jeho rozšířením jej lze najít i v neobvyklých projektech, kde ani jiné databáze dost dobře použít nelze, jako jsou třeba geografické aplikace.

Platforma .NET společnosti Microsoft se pomalu stává primárním způsobem vývoje aplikací pro operační systém Windows a to jak pro serverové, tak klientské aplikace. Díky standardizaci lze ale .NET najít i na Linuxu v podobě projektu Mono. Z této stále ještě relativně mladé technologie se pomalu stává jeden z hlavních hráčů v oblasti vývoje software.

Vzhledem k předchozím odstavcům se lze ptát, jak na tom spolupráce těchto dvou technologií, zda a jak je možné je propojit a zda je možné pracovat s databází PostgreSQL z .NET Frameworku. To je také jedna z otázek, kterou si klade tato práce, když hledá nejvhodnější řešení přístupu k PostgreSQL z .NETu.

Svět kolem nás se skládá z různých objektů, od své přirozenosti jsme schopni a máme potřebu vnímat věci kolem sebe jako objekty. A tento způsob vnímání se přenáší i do informačních technologií, především pak v podobě objektově orientovaného programování. Pro ukládání dat v databázích se však osvědčil relační přístup, který je od objektového pojímání světa značně vzdálen. Proto vznikají různé frameworky pro tzv. objektově-relační mapování, které umožňuje reprezentovat relační data v podobě objektů. Dalším dílčím úkolem této práce je vytvořit takovýto jednoduchý framework, který by mapoval tabulky v databázi PostgreSQL do objektů prostředí .NET.

Některé aplikace už ze své podstaty obsahují rozsáhlé množství dat. Při požadavku na jejich zobrazení se můžeme do problému, že jsme od datového zdroje k aplikaci nuceni přenášet všechna tato data, i když pro vlastní prezentaci uživateli se jich použije třeba jen nepatrný zlomek. Navrhnout vhodnou komponentu v podobě gridu, která by řešila efektivní zobrazování velkého počtu dat, je posledním z hlavních úkolů, které tato práce řeší.

1 Teoretický úvod

Chceme-li se zabývat metodami přístupu k databázím PostgreSQL z .NET frameworku, je nezbytně nutné seznámit se nejprve se souvisejícími technologiemi. Těch není zrovna malé množství, neboť daná problematika má značný rozsah počínaje databázovými systémy a prací s nimi, přes specifika konkrétní databáze PostgreSQL, až k problematice samotné platformy .NET. Ta samotná nabízí celé spektrum věcí, které si je třeba ujasnit – její architekturu a základní principy, možnost práce s daty v podobě frameworku ADO.NET a v neposlední řadě také souvislost s programovacím jazykem C#.

Základní popis těchto jmenovaných technologií je cílem této první kapitoly.

1.1 Databáze

Většina počítačových aplikací pracuje s nějakými daty, která mohou být ukládána v různých podobách. Mnohdy si vystačíme s jednoduchou formou např. v podobě textového souboru. Častěji však ukládanými daty popisujeme reálný svět kolem nás. K jeho popisu si už s jednoduchou formou nevystačíme, potřebujeme nějaký komplexnější systém. A zde získáme prostor, kdy můžeme hovořit o databázi.

Pojem *databáze* pochází původně z anglického *data base* neboli *báze dat* a označoval pouze vlastní uložená data. V praxi se však často používá pro označení celého *databázového systému*.

1.1.1 Databázový systém

Databázovým systémem označujeme spojení dvou pojmů – databáze a Systému řízení báze dat (SŘBD), anglicky Database Management Systems (DBMS).

Zatímco první pojem představuje strukturovaně uloženou kolekci vzájemně souvisejících dat, druhý označuje množinu knihoven, aplikací a nástrojů umožňujících správu uložených dat.

Mezi základní služby, které databázový systém poskytuje, patří například [1]:

- Ukládání dat do fyzických souborů a jejich opětovné získávání.
- Správa současného přístupu více uživatelů k datům.
- Správa transakcí, které znamenají současné vykonání několika změn v databázi v rámci jedné nedělitelné jednotky.
- Podpora dotazovacího jazyka.
- Mechanizmy pro zálohování databáze a pro zotavení po havárii.
- Bezpečnostní mechanismy pro zabránění neoprávněnému přístupu k datům a jejich neoprávněné modifikaci.

1.1.2 Modely databází

Data jsou do databází ukládána na základě určité architektury, která se snaží odrážet podobu reálného světa. Během mnoha let vývoje databázových systémů vzniklo několik základních modelů. [1]

Otevřené soubory

Otevřené soubory (flat files) jsou obyčejné soubory operačního systému, které nenesou žádnou informaci o struktuře uložených dat či nějakém vztahu mezi jednotlivými záznamy, tyto informace musí být součástí příslušné aplikace.

Otevřené soubory v podstatě nejsou databázemi v pravém slova smyslu, protože nesplňují řadu pro databáze nezbytných podmínek. Přesto zde mají své místo, neboť existovaly dávno před vznikem databází a první databázové systémy se vyvinuly právě z nich.

Mezi typické představitele otevřených souborů patří například soubory `.ini` a `.csv` či soubor `/etc/passwd` pro ukládání informací o uživatelích v systémech rodiny Unix.

Hierarchický model

Hierarchický model se vyvinul z původních souborových systémů a záznamy v něm byly uspořádány do hierarchie podobné třeba organizačnímu diagramu. Každý ze souborů v systému otevřených souborů je zde nahrazen *typem záznamu* neboli *uzlem*, které jsou propojeny pomocí *ukazatelů*, které obsahují adresu příslušného svázaného záznamu. Ukazatel definuje vztah rodič-potomek, kdy rodič může mít více potomků, ale potomek pouze jednoho rodiče (vztah *jedna k více*).

Nejrozšířenější hierarchickou databází býval systém ISM (Information Management System) od firmy IBM.

Síťový model

Síťový model vznikl přibližně ve stejné době jako hierarchický model a používá podobné principy (vztah rodič-potomek, propojení pomocí ukazatelů). Na rozdíl od hierarchického modelu jsou zde však vztahy či relace pojmenovány. To dává programátorovi možnost přejít z jednoho záznamu na jiný pomocí konkrétní relace. Jeden typ záznamu se tak na straně potomka může zúčastnit několika relací (vazba více k více). Síťový model umožňuje vyšší flexibilitu ovšem za cenu vyšší složitosti.

Relační model

Síťový a hierarchický model jsou málo flexibilní, poněvadž efektivní zpracování dat je možné pouze při průchodu po předem definované cestě. Pro zpracování jednorázových (ad hoc) dotazů je potřeba prohledat celou databázi.

Relační model je postaven na matematickém aparátu relačních množin a predikátové logiky, založeném na výzkumné práci Dr. E. F. Codda publikované v červnu 1970. Data jsou reprezentována pomocí dvourozměrných tabulek. Tyto tabulky nazýváme relacemi. Tabulky je možné spojovat do pohledů, které jsou opět dvourozměrnou tabulkou, tedy relací.

Relační model umožňuje na rozdíl od hierarchického či síťového modelu svazovat záznamy podle potřeby, navíc dotazy pracují vždy s určitou množinou dat, nikoliv jednotlivými záznamy.

Objektově orientovaný model

Důvodem ke vzniku tohoto modelu v průběhu sedmdesátých let bylo, že tehdejší relační databáze nebyly schopné ukládat složitější datové typy, jako jsou například obrázky či zvukové soubory.

Objektově orientovaný model vychází z principů objektově orientovaného programování, kdy pod pojmem objekt rozumíme seskupení dat a programové logiky, které společně reprezentuje nějakou věc reálného světa.

Nejdůležitějším rozdílem oproti ostatním modelům je, že k datům můžeme přistupovat pouze prostřednictvím *metod* daného objektu. Této vlastnosti říkáme *zapouzdření* objektů.

Objektově relační model

Objektově orientovaný model přináší díky zapouzdření určité výhody, nicméně kvůli chybějící možnosti vytvářet jednorázové (ad hoc) dotazy, nalézá uplatnění jen v omezených, specifických oblastech.

Někteří výrobci relačních databázových systémů se pokusili využít to nejlepší z obou modelů a doplnili objektově orientované funkce do svých databázových produktů, pro které se uchytil pojem objektově relační databáze.

1.2 Jazyk SQL

Když se v 70. letech 20. století začala rozvíjet myšlenka relačních databází, rozhořely se vášnivé diskuze mezi přívrženci nového relačního modelu a zastánci starších síťových a hierarchických modelů. Jedním z argumentů kritiků nového modelu bylo, že relační model je příliš matematický. To vedlo k vytvoření dotazovacích jazyků, které umožňovaly odstínění od složitých matematických problémů. [1]

Jedním z prvních, a pro nás nejvýznamnějších, dotazovacích jazyků byl *SEQUEL*, což je akronym pro *Structured English Query Language*, který byl poprvé použit v databázi System-R firmy IBM. Systém Ingres, předchůdce PostreSQL, vyvíjený paralelně se System-R na University of California at Berkeley (UC-Berkeley) používal svůj vlastní dotazovací jazyk označovaný jako QUEL. Standardem pro

dotazovací jazyky se nakonec stal SEQUEL, který byl přejmenován na *Structured Query Language (SQL).* [2]

První databází na trhu používající SQL byl Oracle a bylo to v roce 1980. Důležitým rokem pro SQL byl rok 1986, kdy bylo SQL standardizováno organizací American National Standards (ANSI) a o rok později byl standard přijat organizací International Organization for Standardization (ISO). Následovalo několik dalších verzí, z nichž do současné doby zůstávají nejdůležitější SQL-92 z roku 1992 a SQL:1999 z roku 1999. Další verze (SQL:2003 a SQL:2006) upravují především práci s XML v databázích.

1.2.1 Části jazyka SQL

Jazyk SQL lze rozdělit do čtyř základních částí podle toho, k čemu slouží. [1]

Jazyk pro dotazování

Pod označením jazyk pro dotazování (Data Query Language, DQL) se v podstatě skrývá jediný příkaz, a to příkaz `SELECT`. S jeho pomocí můžeme získávat záznamy z jedné či více databázových tabulek.

Jazyk pro manipulaci s daty

Jazyk pro manipulaci s daty, anglicky Data Manipulation Language (DML), představuje příkazy, pomocí kterých můžeme přidávat, upravovat a odstraňovat záznamy v databázi. Patří sem příkazy `INSERT`, `UPDATE` a `DELETE`.

Jazyk pro definici dat

Jazyk pro definici dat, anglicky Data Definition Language (DDL), v sobě zahrnuje příkazy `CREATE`, `DROP` a `ALTER`. Jejich pomocí můžeme vytvářet, odstraňovat a upravovat databázové objekty zadaného typu (tabulka, sloupec, funkce,…).

Jazyk pro řízení dat

Příkazy `GRANT` a `REVOKE`, skrývající se pod označením jazyk pro řízení dat (Data Control Language, DCL), slouží k nastavení systémových oprávnění a přístupových práv k objektům.

1.3 Databáze PostgreSQL

PostgreSQL je *objektově relační databázový systém s otevřeným zdrojovým kódem* (open source) s pověstí spolehlivého systému s dobrou integritou dat. Lze jej provozovat na všech hlavních operačních systémech, tedy Linuxu, Unixu i Windows. [3]

1.3.1 Základní vlastnosti PostgreSQL

PostgreSQL má plnou podporu pro cizí klíče (foreing keys), spojování tabulek (joins), pohledy (views), spouště (triggers) a uložené procedury (stored procedures) v mnoha jazycích. Obsahuje většinu datových typů standardů SQL92 a

SQL99, např. *integer*, *numeric*, *boolean*, *char*, *varchar*, *date*, *interval* a *timestamp*. Má podporu pro ukládání binárních objektů jako jsou obrázky, zvuky či videa. [3]

PostgreSQL plně splňuje podmínky **ACID** (*Atomic*, *Consistent*, *Isolated*, *Durable*), což je obecně uznávaný seznam požadavků na bezpečný transakční systém. *Atomičnost* označuje, že v rámci transakce jsou provedeny všechny změny nebo žádná, *konzistence*, že transakce zajišťují převedení dat z jednoho konzistentního stavu do druhého, *izolace*, že transakce není ovlivněna souběžnými transakcemi, a konečně *trvanlivost*, že pokud je transakce potvrzena, pak jsou změny dat trvalé a to i pokud nastane havárie systému. [4]

Značnou výhodou PostgreSQL je jeho **rozšiřitelnost**. Uživatelé jej mohou rozšiřovat například o vlastní datové typy, funkce, operátory, agregační funkce, indexační metody a dokonce celé procedurální jazyky. V PostgreSQL tak není problém vytvořit datové typy pro geometrická a prostorová primitiva či třeba síťové adresy. Díky dobré modularitě PostgreSQL mohou vznikat různá rozšíření, jako například *PostGIS* pro podporu geografických informačních systémů či Slony-I pro asynchronní replikaci databází.

PostgreSQL podporuje národní znakové sady, vícebajtové kódování a Unicode, bere ohledy na národní specifika pro řazení, formátování a citlivost na velikost znaků (case-sensitivity). Case-sensitivní je také pro názvy např. tabulek či sloupců, pokud jsou uvedeny v uvozovkách (zatímco `foo`, `Foo` i `"foo"` odkazují na jednu a tu samou tabulku, zápis `"Foo"` označuje tabulku jinou). Toto chování neodpovídá SQL standardu. Požadujeme-li přenositelnost aplikací, měli bychom názvy uvozovat buď vždy nebo naopak nikdy. [3] [5]

PostgreSQL je vysoce škálovatelné co do množství dat, které může spravovat, tak počtu uživatelů, kteří k datům přistupují. Některé obecné limity PostgreSQL jsou uvedeny v tabulce Tabulka 1.1.

Tabulka 1.1: Obecné limity PostgreSQL

Limit	**Hodnota**
Maximální velikost databáze	Neomezená
Maximální velikost tabulky	32 TB
Maximální velikost řádku	1,6 TB
Maximální velikost pole	1 GB
Maximální počet řádků na tabulku	Neomezen
Maximální počet sloupců na tabulku	250 až 1600 v závislosti na typu sloupců
Maximální počet indexů na tabulku	Neomezen

1.3.2 Procedurální jazyk PL/pgSQL

PostgreSQL podporuje celou řadu programovacích jazyků, které lze použít pro psaní funkcí, uložených procedur. V první řadě je to samotný jazyk C, dále pak řada jazyků jako Perl, Python, PHP či Java. Významnou roli při vývoji aplikací s databázemi PostgreSQL pak hraje procedurální programovací jazyk *Procedural Language/PostgreSQL Structured Query Language (PL/pgSQL)*.

PL/pgSQL je plnohodnotný programovací jazyk umožňující mnohem více procedurální řízení než samotné SQL, jako jsou například smyčky a další řídící struktury. Je velice podobný jazyku PL/SQL firmy Oracle a funkce jím vytvořené mohou být volány z SQL příkazů i jako akce, co vykonávají triggery.

SQL mohou být na serveru vykonány pouze po jednom. To znamená, že v případě nějaké komplexnější operace musí aplikace poslat požadavek databázovému serveru, čekat na zpracování, přijmout a zpracovat výsledek a následně rozhodnout o dalším dotazu na server. Toto zbytečně zvyšuje komunikaci mezi aplikací a serverem. S PL/pgSQL můžeme seskupovat bloky výpočtů a skupiny příkazů uvnitř databázového serveru, a to za využití výkonu procedurálního jazyka a jednoduchosti SQL, a tím tuto komunikaci snížit. [3]

Následující kód uvádí příklad jednoduché funkce pro získání kurzoru v jazyce PL/pgSQL:

```
CREATE OR REPLACE FUNCTION get_products(cur_name refcursor)
RETURNS refcursor AS
$$
BEGIN
  OPEN $1 FOR SELECT * FROM products;
  RETURN $1;
END;
$$
LANGUAGE plpgsql;
```

1.3.3 Stručná historie PostgreSQL

Předchůdcem systému PostgreSQL byl systém Ingres vyvíjený na kalifornské univerzitě v Berkeley mezi lety 1977 a 1985. Na jeho úspěch navázal v roce 1986 prof. Michael Stonebraker, který projekt dále rozvíjel pod označením Postgres. Po několika letech vývoje několik studentů Dr. Stonebrakera upravilo Postgres pro podporu SQL (dříve Postgres používal svůj vlastní dotazovací jazyk) a v roce 1995 byl Postgres uvolněn jako Postgres95. Vzrůstající obliba SQL pomohla Postgres dostat se mezi hlavní databázové produkty. O rok později opustil Postgres prostředí univerzity v Berkeley jako projekt s otevřeným kódem a na základě nedávno přidané podpory pro SQL byl přejmenován na PostgreSQL.

První verze PostgreSQL uvolněná v roce 1996 jako open source nesla označení 6.0. V současnosti (květen 2009) se PostgreSQL nachází ve verzi 8.3.7 a pracuje se na verzi 8.4.

Podstatnou změnu ve vývoji PostgreSQL znamenala verze 8.0 z roku 2005, která byla jako první v historii nativně určena i pro operační systém Windows (do té doby bylo možné spustit PostgreSQL pod Windows pouze prostřednictvím Cygwin). Důležitým krokem byla rovněž verze 8.3, která umožňuje celý projekt zkompilovat pomocí Microsoft Visual C++. [6] [5]

1.3.4 Výhody a nevýhody PostgreSQL

Velmi významnou výhodou PostgreSQL je velmi liberální open source licence (BSD), která umožňuje libovolné rozšiřování a úpravy databáze a to i pro komerční účely. Ve srovnání s komerčními produkty za těmito nikterak nezaostává, naopak mnohdy je předčí především díky své rozšiřitelnosti.

Nepřehlédnutelným plusem je velice dobrá dokumentace celého projektu, která umožňuje uživatelům v tak rozsáhlém projektu, jakým PostgreSQL je, dobrou orientaci.

Za jediný největší nedostatek PostgreSQL lze označit snad jen jeho malou rozšířenost mezi běžnými uživateli a nízkou podporu v našich krajích např. na webových serverech.

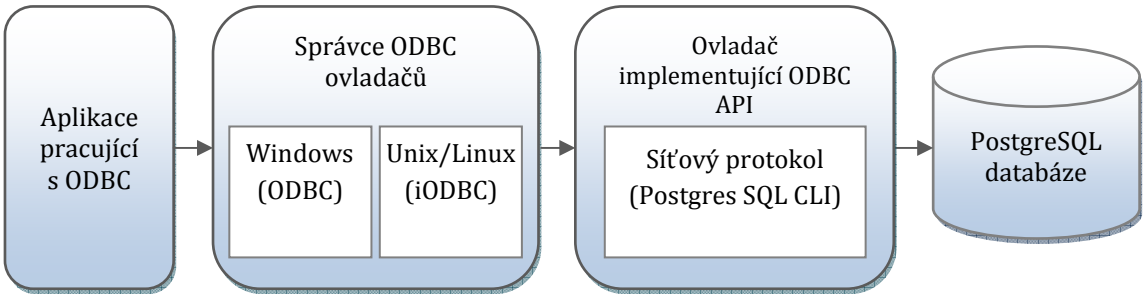
1.4 Techniky přístupu k datům

Existuje velké množství databázových systémů a jiných datových zdrojů, které mají každý své specifické vlastnosti. I když byl pro dotazování nad databázemi vyvinut společný dotazovací jazyk SQL, který by měl komunikaci sjednocovat, přesto u jednotlivých systémů existují drobné nuance v jeho implementaci a především v tom, jak vůbec SQL příkazy systému zadávat, jak s ním komunikovat. Proto v průběhu času vzniklo několik různých technik, jak se s těmito rozdíly vypořádat.

1.4.1 ODBC

Open Database Connectivity (ODBC) byla první technologií, které se snažila o poskytnutí univerzálního rozhraní pro přístup k datům nezávisle na platformě.

Počátkem devadesátých let 20. století začala skupina softwarových společností SQL Access Group (SAG) pracovat na specifikaci rozhraní pro programování aplikací (Application Programming Interface, API), které by definovalo, jak mají aplikace posílat SQL příkazy databázovému systému a jak konzistentně pracovat s navrácenými daty. Toto rozhraní bylo původně označeno jako Microsoft's ODBC API, později, v roce 1993, bylo standardizováno jako *Call Level Interface (CLI)* a je dodatkem SQL-92. Ke konci roku 1994 převzala kontrolu nad specifikací ISO SQL/CLI společnost X/Open, která standard významně aktualizovala a rozšířila. X/Open CLI je nadmnožinou ISO SQL/CLI.



Obrázek 1-1: Příklad architektury pro přístup k databázi PostgreSQL pomocí rozhraní ODBC

ODBC je implementováno do většiny operačních systémů a existují pro něj stovky ovladačů zahrnující také všechny významné databázové systémy jako Oracle, Microsoft SQL Server, Sybase, MySQL, PostgreSQL a mnoho dalších. Mezi nejznámější implementace CLI patří Microsoft ODBC a iODBC.

V operačních systémech Windows je ODBC součástí frameworku *Microsoft Data Access Components (MDAC)* a je definováno jako *Dynamic Link Library (DLL).* Pro přístup ke konkrétní databázi je potřeba mít nainstalován specifický ovladač, který s ní umí komunikovat.

Správce ODBC ovladačů je nepatrná vrstva řídící komunikaci mezi aplikací a ODBC ovladačem, s kterým aplikace pracuje. Je zodpovědná za načtení požadovaného ovladače.

ODBC ovladač odpovídá na všechna volání ODBC API z aplikace. V případě, že dostane SQL požadavek obsahující SQL syntaxi, které cílový databázový server nerozumí, přeloží jej do srozumitelné formy. [7] [8]

1.4.2 OLE DB

Technologie **Object Linking and Embedding Database (OLE DB)** je dalším z pokusů firmy Microsoft získat nástroj pro sjednocený přístup k různým druhům dat. Vznikl v dobách největší slávy Component Object Model (COM) a poskytuje objektově orientované API nad různé relační databáze, ale i nerelační zdroje dat, jako jsou objektové databáze nebo soubory tabulkových procesorů (např. Microsoft Excel).

OLE DB bylo navrženo jako náhrada pro ODBC. Nicméně zatímco ODBC bylo navrženo pro poskytování přístupu primárně k SQL datům v multiplatformním prostředí, OLE DB má poskytovat přístup ke všem datům v OLE Component Object Model prostředí.

OLE DB poskytovatelé (providers) mohou být rozděleni jako poskytovatelé dat a služeb. Poskytovatel dat je ten, kdo vlastní data a poskytuje je ve formě tabulek (databáze, tabulkový soubor). Poskytovatelem služeb se nazývá ta část OLE DB, která nemá vlastní data, ale poskytuje některé služby pro produkování či spotřebovávání dat přes rozhraní OLE DB (zpracování dotazů, správa transakcí).

OLE DB je v operačních systémec Windows stejně jako ODBC součástí Microsoft Data Access Components.

1.4.3 ADO

Programátoři používající jazyky podporující ukazatele (C, C++,…) mohli komunikovat přímo s ODBC a OLE DB API, avšak programátoři používající jazyky jako např. Visual Basic potřebovali další vrstvu. Postupně vzniklo několik technologií jako *Data Access Objects (DAO)* či *Remote Data Objects (RDO)*. Obě tyto byly nakonec nahrazeny technologií novou, nazvanou **ActiveX Data Objects (ADO)**.

ADO bylo navrženo jako nástavba pro OLE DB a umožňuje tedy přistupovat k libovolným datovým zdrojům. Zároveň přineslo čistší objektový model a řadu nových možností, jak vyvíjet aplikace. Patří sem především tzv. *Batch Updating*, který umožňuje provést změny dat v celém objektu Recordset v paměti a tyto změny pak přenést zpět do databáze, dále možnost pracovat s daty v odpojeném stavu a v neposlední řadě podpora pro vícenásobný Recorset. [9]

1.4.4 ADO.NET

ADO.NET je technologií pro přístup k datům v .NET Frameworku, tedy z aplikací s řízeným kódem. Ačkoliv nese v názvu označení ADO a je určitým způsobem následovníkem ADO, mnoho společného s touto technologií ale nemá.

Podrobněji se popisu ADO.NET věnuje kapitola 1.7.

1.5 .NET Framework

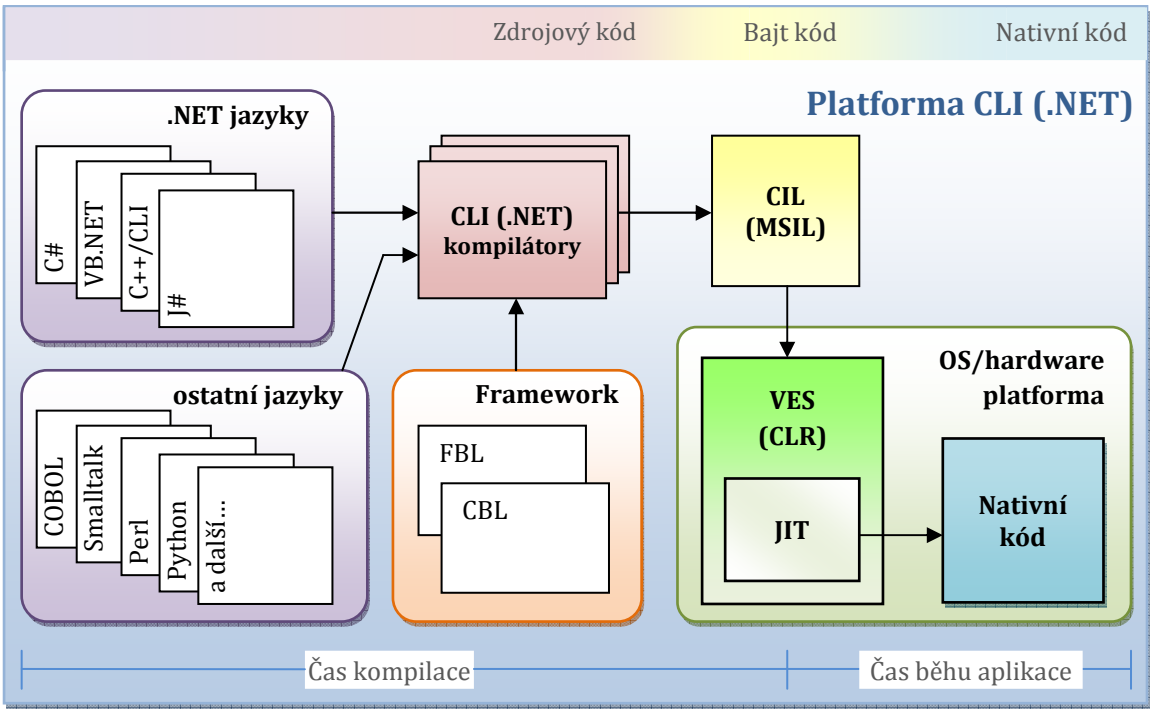
V polovině 90. let 20. století uvedla firma Sun Microsystems programovací jazyk Java umožňující díky virtuálnímu stroji Java Virtual Machine (JVM) psát multiplatformní aplikace. Java si rychle získala přízeň programátorů a možná že právě to bylo podnětem pro Microsoft, aby začal s přípravou vlastního řešení, které je dnes známé jako Microsoft .NET Framework, či jen krátce .NET.

O .NET můžeme mluvit jako o platformě, která sestává z mnoha dílčích technologií, z nichž některé byly později standardizovány jako standardy ECMA-335 a ISO/IEC 23271:2006. Tato standardizovaná verze .NET se nazývá Common Language Infrastructure (CLI) s specifikuje, jak má vypadat programový kód aplikací a prostředí, v kterém je tento kód vykonáván.

Vedle Microsoft .NET existují i další projekty, které jsou postaveny na základě specifikace CLI, a to například projekt Mono či DotGNU.

Standard CLI definuje specifikace těchto svých částí:

- Virtual Execution System (VES)
- Common Intermediate Language (CIL)
- Common Type System (CTS)
- Common Language Specification (CLS)
- Framework



Obrázek 1-2: Struktura platformy CLI. Údaje v závorkách označují pojmy používané v prostředí Microsoft .NET.

1.5.1 Common Type System (CTS)

CTS stojí v samotném centru CLI a tvoří jednotný typový systém, který je sdílen kompilátory, nástroji i CLI samotným. Pro systémy postavené na specifikaci CLI definuje pravidla pro deklaraci, používání a správu datových typů. [10]

CTS umožňuje, aby komponenty či datové typy definované v jednom programovacím jazyce mohly být použity i v jazyce jiném. Díky tomu je možné psát aplikace pro .NET (resp. CLI) v mnoha různých jazycích.

CTS definuje dva druhy datových typů – hodnotové (např. `int`, `float`, `enum`) a referenční (objekty, pole, řetězce, delegáty,…) a vytváří jejich hierarchickou strukturu, umožňuje provádět jejich verifikaci a zabezpečuje typovou bezpečnost. CTS vynucuje, aby všechny třídy byly odvozeny od třídy System.Object. [10] [11] [12]

1.5.2 Common Language Specification (CLS)

CLS je v podstatě podmnožinou CTS. Definuje soubor syntaktických a sémantických pravidel a standardů, kterým musí vyhovovat všechny .NET kompatibilní vývojové nástroje a jejich kompilátory [11].

1.5.3 Common Intermediate Language (CIL)

Common Intermediate Language – v době před standardizací CLI nazývaný také jako Microsoft Intermediate Language (MSIL) – je nízko úrovňový (low-level) a lidem srozumitelný (human-readable) programovací jazyk s objektovými prvky vyšších jazyků.

Zatím co kompilátory pro C, C++ a další ne.NET jazyky generují přímo strojový kód pro konkrétní typ procesoru, všechny .NET-kompatibilní překladače generují tzv. bajt kód v mezijazyce CIL. Důsledkem je, že kompilací zdrojového kódu napsaného v libovolném jazyce (C#, VB.NET, C++/CLI, COBOL, Smalltalk, Perl, Python, Ruby, PHP,…) získáme vždy stejný, platformě nezávislý a tedy **přenositelný kód** programu.

1.5.4 Virtual Execution System (VES)

Tento systém implementuje a vynucuje CTS model a je zodpovědný za načtení a běh programů napsaných pro CLI. Poskytuje služby potřebné k provádění řízeného kódu a dat. [10]

VES bývá označován jako běhové prostředí či runtime, v prostředí .NET se nazývá **Common Language Runtime (CLR)**.

V okamžiku požadavku na spuštění .NET aplikace je operačním systémem vytvořen samostatný proces, do něhož je načteno společné běhové prostředí CLR. Poté, co je běhové prostředí inicializováno, dochází k načtení **Just-In-Time (JIT)** kompilátoru, který následně provede překlad CIL kódu do nativního strojového kódu. Nepřekládá se však celý CIL kód, nýbrž jen ta jeho část, která je v danou chvíli nezbytně nutná pro běh aplikace. I po překladu je .NET aplikace plně pod kontrolou CLR, který ji řídí, spravuje. Proto mluvíme o řízených aplikacích a **řízeném kódu** (managed code).

Výhodou takovéhoto řízeného kódu je, že mu běhové prostředí poskytuje vysoký komfort. Poskytuje mu službu Garbage Collection a celkově se stará o správu paměti, stará se o zpracování výjimek, řízení vláken, vynucování bezpečnostních pravidel a další věci, které ve výsledku ulehčují práci vývojářům. [11] [13] [12]

1.5.5 Assembly

Assembly neboli sestavení aplikace má zpravidla podobu spustitelného souboru (.exe) či dynamicky linkované knihovny (.dll) a jeho nejjednodušší podoba se skládá ze čtyř částí [11]:

Manifest

Obsahuje metadata popisující sestavení. Nese informace o názvu, číslu verze a kultuře sestavení, dále obsahuje odkazy na jiná sestavení a seznam všech souborů, které do dané assembly patří.

Typová metadata

Assembly si s sebou nesou relevantní informace o definovaných datových typech, čehož lze s výhodou použít pomocí mechanizmu reflexe.

Programový kód jazyka CIL

Vlastní seznam instrukcí programu v jazyce CIL.

Zdroje

Může se jednat o grafické soubory, ikony, textové řetězce a další zdroje, které lze použít například při stavbě grafického uživatelského rozhraní aplikací.

1.5.6 Framework

CLI specifikuje množinu základních tříd, které jsou nezbytné k vykonání .NET aplikací. Tyto třídy jsou označovány jako **Base Class Library (BCL)**. Tato knihovna nabízí náhradu za mnoho současných API, a i když jsou Win32 API stále přístupná, ve většině případů lze zvolit třídy z BCL, které vyhovují koncepci objektově orientovaného programování (OOP). [12]

Mimo BCL existuje velké množství dalších tříd označovaných jako **Framework Class Libraries (FCL)**. Ty obsahují API pro tvorbu grafických uživatelských rozhraní (WinForms), databázové aplikace (ADO.NET), webové služby (ASP.NET) a další.

1.5.7 Vývoj Microsoft .NET Frameworku

První verze .NETu *Microsoft .NET Framework 1.0* byla vydána v roce 2002 a o rok později byla aktualizována na verzi 1.1. V současné době se tyto verze pro vývoj aplikací zpravidla nepoužívají.

Konec roku 2005 přinesl novou, v mnoha ohledech převratnou, verzi označenou jako ***Microsoft .NET Framework 2.0***, jež zahrnovala přepracovaný runtime (CLR) i knihovny BCL a FCL. Součástí byla také nová verze knihoven pro práci s daty ADO.NET 2.0.

Verze **3.0** a **3.5**, které následovaly v listopadu 2006 a 2007 nejsou verzemi v pravém slova smyslu, neboť stále využívají CLR z verze 2.0 a pouze nad ním implementují novější technologie. Z řady rozšíření stojí za zmínku **Language Integrated Query (LINQ)**, jenž byl součástí verze 3.5 a který přináší nový způsob dotazování nad jakýmikoliv daty.

Zatím poslední novinkou byl *.NET Framework 3.5 Service Pack 1* v polovině roku 2008, který kromě zlepšení výkonnosti některých částí frameworku přinesl dvě nové služby pro práci s daty a to *ADO.NET Entity Framework* a *ADO.NET Data Services*.

V druhé polovině roku 2009 by měla vyjít nyní připravovaná verze 4.0, která bude mít přepracované běhové prostředí CRL a přinese lepší podporu pro paralelní programování a také významné změny v oblasti ASP.NET.

1.6 Jazyk C#

C# je objektově orientovaný programovací jazyk vytvořený firmou Microsoft v rámci vývoje .NET platformy. Je jedním z několika desítek programovacích jazyků podporovaných CLR .NET Frameworku. Společně s jazykem Visual Basic .NET je primárním jazykem pro vývoj aplikací v .NET Frameworku. C# je standardizován organizacemi ECMA International (standard ECMA-334) a ISO (ISO/IEC 23270).

Microsoft založil jazyk C# na C++, ale byl silně ovlivněn dalšími jazyky, jako je třeba Java. Právě příslušnost k rodině jazyků C/C++ mu dala jméno. Název C# odkazuje na notu cis (C♯) a má značit, že jazyk C# je o stupeň výše než C/C++ [14].

1.6.1 Rysy jazyka C#

Standard ECMA [15] o jazyce C# uvádí, že to je jednoduchý, moderní, univerzální, objektově-orientovaný programovací jazyk. Poskytuje podporu pro principy softwarového inženýrství, jako jsou přísně typová kontrola, hlídání hranic polí, detekce použití neinicializovaných proměnných a automatický garbage collector. C# Může být použit při vývoji softwarových komponent vhodných pro nasazení v distribuovaných prostředích.

Ačkoliv implementace jazyka C# společnosti Microsoft spoléhá na podporu CLI, jiné implementace, za předpokladu, že dosáhnou základních vlastností požadovaných standardem C#, nemusí. [15]

1.6.2 Vývoj C#

První verze jazyka C# byla vydána společně s .NET Frameworkem 1.0 v roce 2002 a obsahovala základní podporu objektového programování.

Druhá verze (C# 2.0) následovala na konci roku 2005 a obsahovala několik rozšíření zahrnující především generičnost (např. `Dictionary<string, List<int>>` je validní datový typ), dále anonymní metody pro snazší práci s delegáty, částečné třídy, struktury a rozhraní (klíčové slovo `partial` umožňující rozdělit definici např. třídy do více souborů), statické třídy, iterátory a nullable typy (např. `int?` či `bool?`).

V roce 2007 byla s .NET Frameworkem 3.5 a Visual Studiem 2008 vydána verze C# 3.0, která se vyznačuje především zahrnutím LINQ do CRL. Tento krok momentálně není standardizován žádnou organizací.

S rozšířením LINQ souvisí další nové vlastnosti C# 3.0 jako jsou anonymní typy (klíčové slovo `var`) a lambda výrazy (např. `someList.Where(x => x.Size > 5);`). Z dalších vlastní pak stojí za zmínku inicializéry objeků a kolekcí (`Customer c = new Customer { Name="Jack" };`), rozšířené metody (extension methods), částečné metody či automatické vlastnosti.

V současnosti se pracuje na verzi 4.0, která by měla v průběhu roku 2009 přinést podporu pro dynamické programování.

1.7 ADO.NET

ADO.NET je objektový model pro práci s daty, který byl vytvořen jako součást vývoje platformy a frameworku .NET. Umožňuje přistupovat k libovolným datovým zdrojům a zahrnuje v sobě rovněž podporu pro práci s XML. Pro práci s daty poskytuje dva přístupy – odpojený a připojený.

1.7.1 ADO.NET jako následovník ADO

Když začal Microsoft s vývojem .NET, byla nová technologie pro přístup k datům pojmenována jako ADO+. Až později byla shodně s ostaními součástmi .NET Frameworku přejmenována na ADO.NET. Oba tyto názvy odkazují na ADO. Akronym ActiveX Data Object však není zrovna relevantní, poněvadž ADO.NET není ActiveX, jedná se zcela nový model pro přístup k datům. Nicméně Microsoft tuto zkratku ponechal vzhledem k obrovskému úspěchu ADO. [9] [16]

Co je špatného na ADO?

Technologii ADO je v prostředí .NET stále možné použít a to přes *interop* (způsob jak pracovat s COM knihovnami napsanými v neřízeném kódu), ale za cenu ztráty výkonu ve srovnání s plně řízeným kódem. Především však tento způsob neodpovídá .NET zabezpečení.

Špatně navržené COM komponenty mohou chybně spolupracovat s garbage kolektorem a tím může docházet k únikům paměti (memory leaks), což může být v datové vrstvě, jednom z nekritičtějších míst aplikací, velký problém. [17]

Důvody, proč použít ADO

I když jsou .NET aplikace schopné pracovat s ADO, důvodů k tomu, proč tak činit, mnoho není. Tím nejpodstatnějším asi je, že ADO.NET nezahrnuje celou paletu vlastností ADO.

Jedním z největších nedostatků ADO.NET, který nás může donutit sáhnout po ADO, je skutečnost, že ADO.NET nepodporuje práci se serverovými kurzory, které jsou běžnou součástí ADO. [18]

Jak bude popsáno dále, ADO.NET obsahuje pouze dopředný (forward-only) kurzor pro čtení (read-only) v podobě objektu DataReader. Nic jako kurzor s možností zápisu a pohybu vpřed i vzad (scollable) však v ADO.NET není. Verze ADO.NET 2.0 měla ve svém návrhu objekt ResultSet, který by přinesl právě chybějící funkčnost, ale nakonec bylo od jeho zahrnutí do ADO.NET upuštěno. [19]

1.7.2 Poskytovatelé dat ADO.NET

Jedním z klíčových rozdílů mezi ADO a ADO.NET je způsob, jak se staví k práci s různými datovými zdroji. V ADO vždy použijeme stejnou sadu objektů, ať data získáváme z databáze Oracle, SQL Serveru či od jinud. Naopak ADO.NET používá **model poskytovatelů dat**, kdy poskytovatel dat (data provider) je sada tříd určená pro přístup ke konkrétnímu datovému zdroji. Každý tento provider má svoji specifickou implementaci tříd Connection, Command, DataReader a DataAdapter, které jsou optimalizovány pro konkrétní DBMS. [20]

.NET Framework obsahuje čtyři následující poskytovatele:

- **Poskytovatel SQL Serveru** – umožňuje optimalizovaný přístup k databázi SQL Server verze 7.0 a novější
- **Poskytovatel OLE DB** – umožňuje přístup k jakémukoliv zdroji dat, který má nějaký OLE DB ovladač
- **Poskytovatel ODBC** – souží pro přístup ke zdrojům s ovladačem ODBC
- **Poskytovatel Oracle** – pro optimalizovaný přístup k databázi Oracle verze 8i a novější.

Podstatnou vlastností modelu poskytovatelů dat ADO.NET je to, že je rozšiřitelný. To znamená, že pro libovolný zdroj dat si můžeme vytvořit vlastního poskytovatele.

Pokud používáme jiný zdroj dat, než SQL Server či Oracle, je vhodné se nejdříve pokusit sehnat nějakého nativního poskytovatele určeného pro daný zdroj. Takovým je například **Npgsql (.NET data provider for PostgreSQL)**, kterému se budeme věnovat dále v textu. Až pokud žádný poskytovatel pro daný datový zdroj neexistuje, použijeme poskytovatele OLE DB, případně ODBC.

1.7.3 Standardizace v ADO.NET

Na základě informací uvedených v předchozích odstavcích by se mohlo zdát, že ADO.NET nabízí rozdělený model, protože neobsahuje obecnou sadu objektů, které by uměly přistupovat k více databázím.

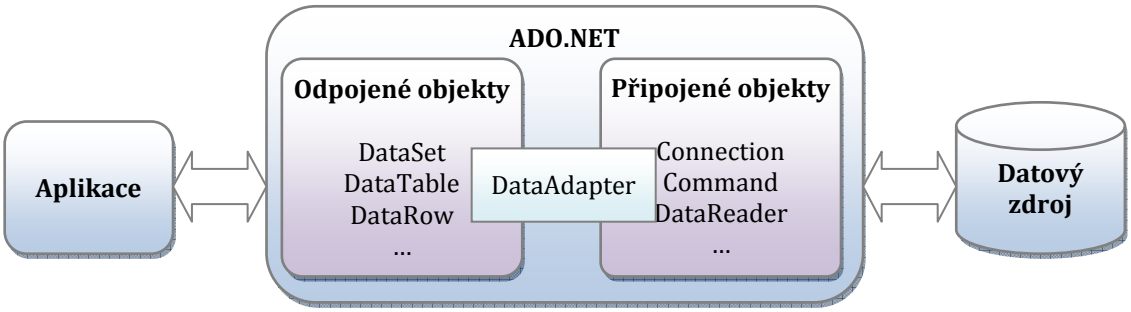
I když různí poskytovatelé dat používají různé třídy, všichni jsou stejně standardizovaní. Všichni jsou totiž založeni na společné sadě rozhraní a tříd, které implementují, resp. používají. Například třídy *SqlConnection*, *OleDbConnection* a *OdbcConnection* implementují jednotné rozhraní *IDbConnection*, které definuje základní společné metody jako `Open()` či `Close()`.

Ačkoliv uvedené třídy používají vnitřně zcela jiné mechanismy – první nativní přístup k SQL Serveru, druhá OLE DB a třetí ODBC ovladač; a obecně mohou používat libovolná nízkoúrovňová volání a API – je zajištěno, že navenek budou vystavovat stejnou sadu metod a vlastností. [20]

V dalším textu, pokud budeme mluvit o obecných vlastnostech ADO.NET, budou namísto názvů konkrétních tříd (SqlCommand, OdbcDataReader,…) používány pseudonázvy bez úvodního označení typu poskytovatele dat (Command, DataReader,…).

1.7.4 Objekty v ADO.NET

Architektura ADO.NET může být rozdělena do dvou základních oblastí: připojené a odpojené. Všechny třídy ADO.NET lze zahrnout do jedné z těchto oblastí. Jedinou velkou výjimku tvoří třída *DataAdapter*, která působí jako prostředník mezi těmito oblastmi.



Obrázek 1-3: Architektura ADO.NET. [17]

Připojené objekty

Připojená část představuje objekty, které pro svoji práci a interakci s datovým zdrojem vyžadují mít k dispozici otevřené připojení.

- `Connection` – umožňuje navázat spojení s datovým zdrojem.
- `Transaction` – umožňuje seskupit několik příkazů a vykonat je se zajištěním atomicity.
- `DataAdapter` – představuje bránu mezi připojenou a odpojenou částí ADO.NET.
- `Command` – umožňuje vykonat příkaz na datovém zdroji. Může nebo nemusí vracet výsledek.
- `Parameter` – umožňuje specifikovat parametry příkazu
- `DataReader` – ekvivalent read-only/forward-only kurzoru umožňující rychlé načítání dat z databáze s velkou rychlostí.

DataReader

Třída *Command* obsahuje, krom jiného, metodu `ExecuteReader()`, která vrací objekt typu *DataReader*. To je, jak již bylo uvedeno, ekvivalent kurzoru k rychlému

načítání dat, která slouží pouze k vypsání (nebudou se tedy zapisovat zpět do databáze).

```
...
DataReader dr = cmd.ExecuteReader();
while(dr.Read())
{
    Console.Write("{0} – {1}", dr['Id'], dr.GetString(1));
}
...
```

Voláním metody `Read()` posunujeme ukazatel záznamů (kurzor) v objektu *DataReader* na další záznam a metoda `Read()` nám vrací booleovskou hodnotu, zda ještě zbývají k přečtení nějaké řádky.

Hodnoty můžeme získávat z objektu *DataReader* buď pomocí indexeru zadáním názvu sloupce (`dr['Id']`) nebo zavoláním metody pro získání hodnoty konkrétního datového typu ze sloupce zadaného jeho pořadovým číslem (`GetString(1)`, `GetInt32(2)`, `GetDateTime(3)`,…).

Odpojené objekty

Neustále připojené aplikace nesplňují požadavky moderních distribuovaných aplikací. Aplikace postavená na ADO.NET mají odlišný přístup. Odpojené aplikace se připojí k datovému zdroji co nejpozději, až když to skutečně potřebují, a opět se odpojí, jak nejdříve to jde.

Jednotlivé objekty odpojeného modelu ADO.NET jsou tyto:

- `DataSet` – základní část odpojeného modelu ADO.NET
- `DataTable`
- `DataRow`
- `DataColumn`
- `DataView`
- `Constraint`
- `DataRelation`

DataSet

DataSet i ostatní uvedené třídy slouží jako jakési „krabice“ na data. Na *DataSet* můžeme pohlížet jako na malou in-memory databázi, lze jej také označit jako off-line kopii datového úložiště. Objekt typu *DataSet* obsahuje objekty *DataTable*, které – jak název napovídá – představují jednotlivé tabulky. Analogicky *DataColumn* a *DataRow* slouží pro definici sloupců a řádků těchto tabulek, *DataRelation* určuje relace mezi jednotlivými tabulkami a *Constraint* určuje omezení jako cizí klíče či unikátní indexy.

Podstatnou vlastností je, že *DataSet* může v jeden okamžik obsahovat tabulky z různých datových zdrojů a je tedy nezávislý na poskytovateli dat a společný všem

poskytovatelům. V rámci CBL jej lze společně s ostatními odpojenými objekty najít ve jmenném prostoru `System.Data`.

Objektu *DataSet* odpovídal v ADO objekt *Recordset.* Nicméně je zde zásadní rozdíl. Zatímco *Recordset* byl také zodpovědný za komunikaci s databází, *DataSet* za ni nezodpovídá. Namísto toho využívá bránu mezi odpojeným a připojeným způsobem – *DataAdapter*. [17]

2 Metody přístupu k PostgreSQL z .NET

Prvním z úkolů této práce je vybrat vhodné řešení, které lze použít pro přístup k databázím PostgreSQL z prostředí .NET Frameworku. Tato kapitola nám poskytne přehled, jak bylo vhodné řešení vybráno a mezi kterými možnostmi byl výběr činěn.

2.1 Přehled možných technologií pro přístup k PostgreSQL

Existuje celá řada programových rozhraní umožňujících psaní klientských aplikací, které mohou komunikovat s databází PostgreSQL. Přímo s PostgreSQL jsou distribuovány dvě z nich – **libpg**, primární rozhraní pro jazyk C, a **ECPG (Embedded SQL in C)**. Přímo v dokumentaci PostgreSQL [5] je pak odkazováno na několik projektů umožňující přístup z dalších jazyků, například **libpqxx** pro C++, **JDBC** pro jazyk Java, **DBD::Pg** pro Perl, **psqlODBC** ovladač pro přístup pomocí rozhraní ODBC či **Npgsql**, poskytovatel pro .NET.

Právě poslední dvě jmenované, tedy psqlODBC a Npgsql, jsou rozhraní, která nás zajímají, pokud chceme k databázi PostgreSQL přistupovat z .NET Frameworku. Na webu *PgFoundry*, PostgreSQL vývojářské skupiny pro vývoj a publikaci software souvisejícího s PostgreSQL, najdeme další projekt – **PgOleDb (PostgreSQL OLE DB provider for Windows)**, který, jak název napovídá, poskytuje rozhraní OLE DB.

Všechny výše uvedené projekty spadají – stejně jako samotné PostgreSQL – do kategorie *open-source*. Pro práci s PostgreSQL ale existují i komerční řešení. V souvislosti s .NET Frameworkem nás mohou zajímat **PostgreSQL Native OLEDB Provider (PGNP)** poskytující OLE DB rozhraní a **dotConnect for PostgreSQL**, poskytovatel určený přímo pro platformu .NET.

Tímto jsme získali pětici různých programových rozhraní, která lze použít pro přístup k databázi PostgreSQL z .NET Frameworku, a v následujícím textu rozebereme jejich vlastnosti.

2.1.1 psqlODBC

psqlODBC [21] je *oficiální ODBC ovladač* pro PostgreSQL. Je napsaný v jazyce C a zdrojové kódy jsou dostupné pod licencí GNU Lesser General Public License (LGPL).

Zdrojové kódy byly původně převzaty z PostODBC verze 0.20. Jako část komerčního výzkumného projektu vývojářů z Insight Distributions System byly přepracovány a výsledky práce zveřejněny. O něco později, a po diskusi s některými členy organizace PostgreSQL, byl psqlODBC upraven jako součást PostgreSQL zdrojové distribuce. Ovladač i nadále udržuje Byron Nikolaidis, vývojář ze společnosti Insight.

psqlODBC je primárně určen pro 32 bitovou platformu Windows a pro tu je také poskytován ve zkompilované binární podobě, nicméně je možné jej přenést a zkompilovat i pod systémy rodiny Unix.

Vývoj psqlODBC reflektuje vývoj samotné databáze PostgreSQL, takže prakticky každá nová verze databáze je následována novou verzí ovladače. V současné době (květen 2009) se tedy psqlODBC nachází ve verzi 08.03.0400.

Po instalaci lze v systému najít dva ovladače – **PostgreSQL Unicode** a **PostgreSQL ANSI**. První z nich je určen pro práci s moderními aplikacemi, které využívají kódování UNICODE, a měl by být použit vždy, pokud pracujeme s databází kódovanou jako UNICODE, resp. UTF-8. Druhý z nich je pak určen především pro práci s daty v některé znakové sadě z rodiny LATIN.

psqlODBC podporuje všechny standardní datové typy PostgreSQL a částečně i ostatní datové typy jako point, circle, box a arrays.

Pro přístup k PostgreSQL s použitím psqlODBC z .NET Frameworku je potřeba použít poskytovatele ODBC, který se nachází ve jmenném prostoru `System.Data.Odbc`.

2.1.2 PgOleDb

PgOleDb, na webu pgFoundry [22] označované jako **PostgreSQL OLE DB provider for Windows**, umožňuje k PostgreSQL přistupovat přes rozhraní OLE DB.

Ovladač je napsán v jazyce C++ a je poskytován pod licencí GNU LGPL. Podle change-logu přiloženého v distribuci započal vývoj někdy počátkem roku 2004, poslední uvolněná verze 1.0.0.20 pochází z dubna 2006 a projekt je stále ve fázi beta. PgOleDb je určeno pro PostgreSQL verze 7.4 a vyšší.

Celkově se projekt jeví jako ne příliš živý a působí zanedbaným dojmem. Neexistuje k němu prakticky žádná dokumentace a ani zdrojový kód neobsahuje téměř žádné komentáře. PgOleDb ani zdaleka nepokrývá celou funkčnost rozhraní OLE DB a podle příspěvků v e-mailové konferenci k projektu [23] obsahuje značný počet chyb.

Toto se potvrdilo i při testování ovladače, kdy při použití objektu *OleDbDataReader* skončil ovladač PgOleDb v blíže nespecifikované chybě. Podle dostupných informací pravděpodobně nedovedl zpracovat data přijatá od databáze po zavolání příkazu `SELECT`.

Z těchto důvodů byl OLE DB ovladač PgOleDb vyřazen z dalších testů a nebude již dále v textu uvažován.

Pro práci s PgOleDb v prostředí .NET je potřeba použít poskytovatele OLE DB, který se nachází ve jmenném prostoru `System.Data.OleDb`. Pro určení ovladače PgOleDb je potřeba nastavit v připojovacím řetězci vlastnost provider na některou z hodnot „PostgreSQL“, „PostgreSQL.1“ nebo „PostgreSQL OLE DB Provider“.

2.1.3 Npgsql

Npgsql [24] je .NET nativní poskytovatel dat pro PostgreSQL, který umožňuje všem programům vyvíjeným pro .NET Framework přistupovat k této databázi. Je zcela napsán **v jazyce C#** (tedy v řízeném kódu) a pracuje s PostgreSQL verze 7.x a 8.x. K dispozici je pod licencí BSD.

Po téměř dvou letech vývoje od Npgsql 1.0 byla v říjnu 2008 vydána verze Npgsql2, která přinesla nové rysy a také podporu pro novější verze .NET frameworku. Npgsql tak nyní vedle .NET 2.0 podporuje .NET 3.5 a díky tomu je možná **podpora ADO.NET Entity Frameworku**. Zároveň je také distribuována verze pro .NET 1.1 a pro platformu Mono.

Změny ve verzi Npgsql2 jsou především na interní úrovni, nejvýznamnější z nich je asi ta, že Npgsql2 nenačítá celý obsah tabulky před tím, než vrátí kontrolu do uživatelského kódu. Výsledkem je mnohem více efektivní zacházení s pamětí při načítání rozsáhlých tabulek.

Npgsql podporuje všechny základní datové typy databáze PostgreSQL i další rozšířené typy a automaticky je přetypovává do objektů prostředí .NET. K tomuto účelu slouží třídy ve jmenném prostoru `NpgsqlTypes`, který je společně s jmenným prostorem `Npgsql` obsahujícím základní třídy jako *NpgsqlConnection* či *NpgsqlDataReader* obsažen v assembly `Npgsql.dll`.

Npgsql je rozsáhlý a fungující projekt poskytující prostředky ke všem běžným úkonům, které můžeme při vývoji běžných aplikací potřebovat. I přes to však „Roadmap“ na webu projektu obsahuje rozsáhlý seznam rysů, které by měly být během budoucího vývoje do Npgsql zabudovány. To dává velkou šanci, že se Npgsql bude i nadále vyvíjet v ještě lepší nástroj.

2.1.4 PGNP

PostgreSQL Native OLEDB Provider (PGNP) [25] poskytuje tenkou vrstvu mezi Microsoft ADO a databází PostgreSQL. Implementuje většinu vlastností OLE DB rozhraní a pro přístup k PostgreSQL používá libpq. PGNP provider může být použit pro 32-bitové a 64-bitové nativní aplikace i aplikace pro .NET. PGNP je napsán v MS C++ (VS 2005) a klíčové funkce jsou implementovány v ručně optimalizovaném assembleru.

PGNP je publikován pod komerční licencí, jedná se o uzavřený kód. Volně dostupná je pouze trial verze produktu, která je omezena pro vrácení nejvýše 100 řádků

tabulky. K této skutečnosti bylo přihlídnuto při testování a testovací kód pracoval právě se 100 záznamy.

Pro použití PGNP v prostředí .NET je potřeba opět použít poskytovatele OLE DB a objektu *OleDbConnection* nastavit vlastnost Provider na hodnotu „PGNP.1“.

2.1.5 dotConnect for PostgreSQL

dotConnect for PostgreSQL [26], dříve známý jako PostgreSQLDirect .NET je rozšířený poskytovatel dat postavený na technologii ADO.NET. Jedná se o komerční řešení společnosti Devart (dříve známé jako Core Lab) používající licenci .NET komponent. Plná verze je tak dostupná pouze za úplatu, nicméně je k dispozici volně použitelná Express verze, která však obsahuje pouze nejzákladnější vlastnosti.

dotConnect for PostgreSQL je napsán zcela v řízeném kódu a podporuje .NET Framework 2.0 i 3.5. Má plnou podporu pro Entity Framework a LINQ, podporu 64-bitových systémů a je kompatibilní s platformou Mono. Podporuje všechny datové typy PostgreSQL včetně složených typů.

Základní třídy pro práci s dotConnect for PostgreSQL se nacházejí ve jmenném prostoru `Devart.Data.PostgreSql`, který je potřeba přidat do projektu používajícího tohoto poskytovatele. Jednotlivé třídy jako *Connection* a další jsou označovány prefixem **PgSql** (*PgSqlConnection*, *PgSqlDataReader*,...). Tento prefix bude díky své jednoduchosti v následujícím textu občas používán k označení celého projektu dotConnect for PostgreSQL.

2.1.6 ADO

Poslední možností přístupu k PostgreSQL z .NET Frameworku, která nebyla zmíněna v úvodu této kapitoly, je použití ADO. Jak bylo uvedeno v kapitole 1.4.3, ADO je pouze jakýmsi obalem nad OLE DB. Z toho vyplývá, že chceme-li z nějakého důvodu použít ADO, budeme potřebovat jedno ze dvou výše uvedených rozhraní OLE DB (PgOleDb a PGNP), případně ODBC (psqlODBC).

2.2 Kritéria pro porovnání

Chceme-li porovnávat uvedené technologie a vybrat z nich tu nejvhodnější pro přístup k PostgreSQL z .NETu, je několik kritérií, které můžeme vzít v úvahu.

Tím nejdůležitějším je, nakolik je daná technologie vhodná pro použití v prostředí řízeného kódu. Jak bylo uvedeno v kapitole 1.7.2, pokud to je možné, je nejlepší vždy použít nativního poskytovatele dat. Tímto se nám výběr hned na začátku pěkně zužuje a zůstává pouze PgSql a Npgsql. Přesto ještě psqlODBC a PGNP chvíli ponechme „ve hře“ a podívejme se, jaká další kritéria se nám nabízí. Jsou to minimálně tyto dvě:

- Porovnání nabízených funkcí

- Porovnání výkonu

V následujícím textu se tedy pokusíme porovnat všechny čtyři technologie – metody přístupu – z těchto dvou pohledů.

2.3 Porovnání nabízených funkcí

Chceme-li porovnávat funkční vybavenost jednotlivých metod přístupu k databázi PostgreSQL, musíme si uvědomit několik základních faktů, které toto porovnání značně omezují.

Předně, v souvislosti s kapitolou 1.7.3 Standardizace v ADO.NET, je to skutečnost, že všechny metody přístupu potřebují nějakého poskytovatele dat, který však nutně musí implementovat jednotná rozhraní ADO.NET. Ať se tedy bude jednat o poskytovatele ODBC pro psqlODBC či OLE DB pro PGNP, nebo samotné nativní poskytovatele Npgsql a PgSql, všichni budou poskytovat stejné metody a vlastnosti a bude se s nimi pracovat stejným způsobem.

Druhým omezujícím faktorem pro srovnávání psqlODBC a PGNP je fakt, že tyto pouze implementují vlastnosti definované návrhem jejich rozhraní a tedy jejich porovnávání by nebylo porovnáním jich samotných, ale spíše porovnáním technologií ODBC a OLE DB.

Z předchozího se dostáváme k závěru, že po stránce funkčnosti, resp. funkční vybavenosti, lze zkusit porovnat pouze Npgsql a PgSql. Vzhledem k tomu, že se jedná o celé poskytovatele dat, kteří jsou zcela napsáni v režii jejich vývojářů, může být množina tříd, které obsahují, širší, než model ADO.NET požaduje. A u obou tomu tak je.

Oba poskytovatelé například shodně obsahují třídy pro podporu speciálních datových typů databáze PostgreSQL, oba také podporují Entity Framework a LINQ. Celkově si však lépe vede dotConnect for PostgreSQL, který nad rámec modelu ADO.NET poskytuje například vlastní upravenou třídu PgSqlDataSet a spoustu dalších. Ty jsou však dostupné pouze v nejdražší komerční verzi dotConnect for PostgreSQL Profesional. Budeme-li hodnotit rozsah volně dostupných funkcí, zde vítězí Npgsql.

2.4 Porovnání výkonu

Při zjišťování výkonu jednotlivých metod můžeme brát v potaz dva hlavní faktory – rychlost a paměťovou náročnost.

Pro porovnání výkonu bylo vybráno pouze porovnání rychlosti. Ne že by nároky na paměť nebyly důležité, nicméně v dnešní době, kdy jsou počítače běžně vybaveny gigabajty operační paměti, ztrácí poněkud na významu. Z pohledu uživatele – a tedy i programátora, vývojáře – je důležitějším faktorem rychlost – čili jak rychlá bude odezva aplikace.

2.4.1 Testovací prostředí

Následující seznam obsahuje hardwarové a softwarové prostředí, v jakém bylo testování provedeno.

- Procesor: Intel Core2 Duo T5250 @ 1,5GHz
- Paměť RAM: 4,0 GB
- Operační systém: Windows Vista Business SP1, 32 bit
- Microsoft .NET Framework 3.5 SP1
- Microsoft Visual Studio 2008 Professional Edition SP1
- PostgreSQL 8.3.5
- PostgreSQL Native Provider 32-bit v1.2.8.1093 (trial verze)
- psqlODBC 08.03.04 (PostgreSQL Unicode)
- Npgsql - .NET Data Provider for PostgreSQL 2.0.1.0 (pro .NET 3.5)
- Devart dotConnect for PostgreSQL Express 4.0.16.0
- Databáze Dell Store 2 (jako testovací data; dostupné [27])

2.4.2 Testovací vzorky

Testování bylo provedeno pro dva vzorky kódu. První z nich používá dopředný kurzor pro čtení v podobě objektu *DataReader*, druhý objekt *DataTable*, který je plněn přes *DataAdapter*.

Uvedené zdrojové kódy používají obecné názvy tříd jako *Connection* či *Command*, v praxi jsou však zastoupeny reálnými třídami jako např. *OleDbConnection* a *NpgsqlCommand.*

```
//--- Testovací kód s využitím objetku DataReader ---
try
{
    // 1
    using (Connection conn = new Connection())
    {
        // 2
        conn.ConnectionString = ConnectionString;
        // 3
        conn.Open();
        // 4
        Command cmd = new Command();
        // 5
        cmd.CommandText = "SELECT * FROM products WHERE prod_id < 100";
        // 6
        cmd.Connection = conn;
        // 7
        DataReader dr;
```

```
            // 8
            using (dr = cmd.ExecuteReader())
            {
                // 9
                while (dr.Read())
                { }
            // 10
            }
        // 11
        }
    }
    catch (Exception)
    {
    }
```

Jednotlivá čísla v komentářích označují místa, ve kterých bylo měřeno, kolik času potřebuje ke svému vykonání kód na následujícím řádku.

```
    //--- Testovací kód s využitím objetku DataTable ---
    try
    {
        // 1
        using (Connection conn = new Connection())
        {
            // 2
            conn.ConnectionString = ConnectionString;
            // 3
            conn.Open();
            // 4
            DataAdapter adp =
               new DataAdapter("SELECT * FROM products WHERE prod_id < 100",
                   conn);
            // 5
            DataTable dt = new DataTable();
            // 6
            adp.Fill(dt);
            // 7
            using (dt)
            {
                // 8
                foreach (DataRow row in dt.Rows)
                { }
            // 9
            }
        // 10
        }
```

```
}
catch (Exception)
{
}
```

2.4.3 Příkaz `using`

U uvedených zdrojových kódů by se mohlo zdát divné, proč měřit čas u složených závorek na konci bloku `using`.

Příkaz `using` představuje pomůcku pro vývojáře, pokud pracují s objekty, které implementují rozhraní *IDisposable*. Zápis `using(IDisposable obj){}` lze totiž přeložit jako

```
try
{
}
finally
{
    obj.Dispose()
}
```

Kód uvnitř bloku `using` odpovídá kódu uvnitř `try`. Ať je uvnitř bloku `using` vyvolána výjimka, nebo kód projde až na konec bloku, vždy je na závěr zavolána metoda `Dispose()`, která uvolní daný objekt z paměti a uvolní rovněž všechny zdroje, které používá. Tuto vlastnost jazyka C# lze s výhodou využít právě při práci s databázemi, kdy se programátor nemusí starat o volání metody `Close()`, ale spojení se serverem je ukončeno automatickým zavoláním metody `Dispose()`.

Právě proto je testován čas strávený nad složenou závorkou, neboť to je **místo, kde se uzavírá spojení s databází**.

2.4.4 Connection pooling

Connection pooling je technika, která umožňuje sdílet jednou fyzicky vytvořené spojení mezi aplikací a databází pro další aplikace. Tím není nutné při každém příkazu `Open()` vytvářet nové spojení, dochází tak ke snížení komunikace mezi aplikací a databázovým serverem, ale především k rychlení aplikace.

Pokud je connection pooling povolený, redukuje se čas potřebný k vykonání metody `Open()` na pouhé vyzvednutí existující připojení z tzv. fondu připojení. Jelikož my chceme naším testem ověřit rychlost jednotlivých metod přístupu k databázi PostgreSQL, tedy i to, jak rychle jsou schopny navázat komunikaci s databází, byl po potřeby testování **connection pooling vypnut**.

2.4.5 Metodika měření

Vykonání uvedených zdrojových kódů je měřeno dvěma způsoby. První, jak již bylo uvedeno, měří čas potřebný k vykonání jednotlivých příkazů. Druhý způsob měří celkový čas potřebný na vykonání celého obsahu bloku `try`.

V testovacích souborech je pak celý uvedený kód umístěn uvnitř cyklu `for`, který v rámci jednoho spuštění testovacího souboru vykoná testovaný kód stokrát. Samotný program pak byl spuštěn celkem čtyřikrát pro každý testovaný soubor. Mezi jednotlivými spuštěními testovacích programů byl vždy restartován databázový server PostgreSQL.

Jak bylo uvedeno v kapitole 1.5, aplikace v prostředí .NET jsou kompilovány do mezijazyka CIL, který je až při spuštění programu překládán do zdrojového kódu počítače. Bylo rovněž uvedeno, že tak není učiněno najednou, ale že se vždy překládá jen ta část, která je nezbytně nutná pro další běh programu. V souvislosti s tímto vykazují hodnoty naměřené pro první vykonání kódu uvnitř testovací smyčky jiné, vyšší hodnoty, než další vykonání téhož kódu. Proto byly tyto hodnoty zaznamenávány odděleně od ostatních naměřených hodnot.

Z předchozích odstavců tedy vyplývá, že byly vykonány testy pro jednotlivé příkazy a pro celý blok kódu a to pro objekty DataReader a DataTable. Pro každou z těchto kombinací bylo získáno 400 hodnot, z čehož čtyři hodnoty představují časy pro první vykonání kódu v rámci spuštění testovací aplikace a zbývající hodnoty představují časy pro druhé a další vykonání kódu. Z těchto dvou skupin hodnot byly vypočítány průměrné hodnoty, které posloužili k porovnání jednotlivých metod.

2.4.6 *Výsledky testování*

Tabulka 2.1: Výsledky pro první vykonání kódu s objektem DataReader

Operace	Průměrný čas vykonání operace [µs]			
	Npgsql	PGNP	PgSql	psqlODBC
1	127 791,03	4 079,62	2 459,29	4 063,12
2	15 218,34	1 187,55	19 724,70	1 075,73
3	250 745,07	378 420,34	206 436,82	341 896,54
4	1 306,66	9,10	262,95	6,11
5	197,02	19,47	1 849,08	19,73
6	619,70	11,89	147,28	11,87
7	1,47	3,09	1,43	3,63
8	6 138,14	170 987,58	29 813,06	405 563,82
9	645,82	463,94	6,06	1 136 783,64
10	6,84	195,00	1 530,55	9 287,86
11	3 910,69	1 183,48	2 988,33	27 495,87

Tabulka 2.2: Výsledky pro druhé a další vykonání kódu s objektem DataReader

Operace	Průměrný čas vykonání operace [µs]			
	Npgsql	PGNP	PgSql	psqlODBC
1	284,81	9,81	5,92	10,70
2	318,35	13,15	68,92	14,88
3	56 221,93	86 213,00	60 119,93	182 255,79
4	107,41	8,02	3,78	5,82
5	3,57	4,63	2,21	5,25
6	3,32	3,81	1,60	5,84
7	3,09	2,90	1,49	3,20
8	4 441,43	30 277,67	4 795,66	423 179,47
9	261,79	271,96	6,11	1 146 093,19
10	6,34	46,95	5,94	9 642,45
11	150,18	558,64	120,97	28 730,47

Tabulka 2.3: Výsledky pro první vykonání kódu s objektem DataTable

Operace	Průměrný čas vykonání operace [µs]			
	Npgsql	PGNP	PgSql	psqlODBC
1	43065,25	4044,33	2559,60	4466,63
2	14919,21	1159,64	20115,91	1183,09
3	235150,49	246246,85	221322,37	317400,10
4	1903,87	4037,98	3636,15	43,21
5	73,54	82,67	1046,45	79,51
6	10569,71	46876,17	47283,28	5320053,28
7	1,61	1,50	1,48	1,48
8	73,61	78,22	79,88	90,27
9	20,46	36,91	21,11	22,21
10	3800,06	2847,94	4589,55	30781,50

Tabulka 2.4: Výsledky pro druhé a další vykonání kódu s objektem DataTable

Operace	Průměrný čas vykonání operace [µs]			
	Npgsql	PGNP	PgSql	psqlODBC
1	295,46	9,61	5,83	8,25
2	329,31	13,32	68,80	10,48
3	55780,68	83039,70	60716,51	184720,21
4	103,71	10,50	7,27	6,88
5	15,11	20,49	16,19	16,32
6	5413,62	32646,77	5575,29	5532407,99
7	1,58	1,58	1,50	1,50
8	19,03	18,04	15,32	24,86
9	6,96	6,25	2,94	5,11
10	159,58	609,64	156,29	32787,37

Čísla operací odpovídají jednotlivým řádkům testovacího kódu ve shodě s čísly uvedenými ve zdrojových kódech v kapitole 2.4.2.

Tabulka 2.5: Výsledky pro celý blok kódu

	Průměrný čas vykonání bloku kódu [µs]				
	Npgsql	PGNP	PgSql	psqlODBC	ADO (PGNP)
DataReader – první spuštění	315501,6	268314,3	289948,5	1912486	522377,6
DataReader – další spuštění	60392,5	110046,8	65891,99	1804180	110087
DataTable – první spuštění	319203,3	289814,3	290341,6	5792700	-
DataTable – další spuštění	62052,67	112298,5	67917,43	5830260	-

2.4.7 Zhodnocení testů

Všechny naměřené výsledky byly vyneseny do grafů, které lze najít v příloze A . Při pohledu na ně můžeme zjistit mnoho zajímavých věcí.

V první řadě je to potvrzení očekávané skutečnosti, že rozhraní **ODBC je značně pomalé**. Hodnoty naměřené pro ovladač psqlODBC jsou až několikanásobně vyšší než u ostatních metod. Tento fakt způsobil, že při lineárním zobrazení časové osy se rozdíly mezi ostatními metodami stíraly a vynesené hodnoty často splývaly s osou x. Z tohoto důvodu byly naměřené hodnoty vyneseny také v logaritmickém měřítku, které umožňuje zobrazit rozdíly na nižších hodnotách.

Zůstaňme nyní chvíli u výsledků pro testovací kód s objektem ***DataReader*** a dříve, než se budeme věnovat rozdílům mezi jednotlivými metodami, podívejme se na rozdíl mezi prvním vykonáním kódu, při kterém dochází k překladu bajtkódu v jazyce CIL do nativního kódu počítače, a druhým a dalším vykonáním kódu, při kterém už je strojový kód programu k dispozici.

Při pohledu na Obrázek A-2 a Obrázek A-4 je ihned patrný rozdíl, ze kterého je zřejmé, že skutečně dochází k překladu .NET aplikace do nativního kódu. Tyto rozdíly jsou daleko zřetelnější u Npgsql a PgSql, které jsou celé napsané v řízeném kódu a jsou tedy na rozdíl od ovladačů psqlODBC a PGNP rovněž překládány. Toto potvrzuje i Obrázek A-6, který zobrazuje procentuální rozdíl časové náročnosti prvního vykonání kódu vůči hodnotě druhého a dalšího vykonání kódu. Zde je vidět, že v bodech 4, 5, 6, 10 a 11 potřebují Npgsql a PgSql pro první vykonání podstatně vyšší čas, naopak psqlODBC a PGNP se vyjma prvních dvou bodů blíží v celém rozsahu nule. První dva body vykazují zhruba stejné hodnoty pro všechny čtyři metody. To je dáno tím, že ať už se jedná o metodu s řízeným či neřízeným kódem, v obou případech je na začátku potřeba načíst patřičné knihovny a provést operace společné všem metodám.

Procentuální zobrazení, jak jej ukazuje Obrázek A-6, bylo zvoleno z prostého důvodu, neboť daleko přesněji vystihuje to, co chceme vidět – rozdíl mezi prvním a dalším vykonáním kódu. Toho nikdy nemůžeme dosáhnout zobrazením časového rozdílu, jako je na Obrázek A-5, neboť zde bude pro vysoké hodnoty času rozdíl vždy vysoký a pro nízké nízký.

Poslední věcí, kterou lze zmínit v souvislosti s rozdílem mezi prvním a dalším vykonáním kódu, je poněkud nečekaně vysoká časová náročnost u PgSql v bodech 5 (nastavení vlastnosti *CommandText* objektu *PgSqlCommand*) a 10 (zrušení objektu *PgSqlDataReader*). Pravděpodobně dochází ke kompilaci určitých částí kódu, které doposud nebyly použity. Naopak u Npgsql je pravděpodobně větší část kódu zkompilována již v bodě 1 při vytvoření objektu *NpgsqlConnection*, neboť zde je podstatně vyšší časová náročnost než u PgSql.

Nyní se podívejme na rozdíly mezi jednotlivými metodami. Jak již bylo zmíněno, je až do očí bijící malá rychlost rozhraní ODBC. Ovladač psqlODBC je nejpomalejší ve všech *klíčových bodech* jako je navázání spojení (3), vykonání příkazu (8), čtení dat (9) a ukončení spojení s databází (11).

Zbývající tři metody vykazují daleko vyrovnanější výsledky, přesto lze říci, že ovladač OLE DB, především v klíčových bodech, za zbylými dvěma metodami zaostává. PGNP tak v testu rychlosti obsazuje pomyslné třetí místo.

Oba poskytovatelé dat napsaní v řízeném kódu – Npgsql a PgSql – dosahují v klíčových bodech de facto stejných hodnot. Jediný výrazný rozdíl mezi nimi je ve vlastním čtení dat (bod 9), kde je PgSql rychlejší. Objektivně rozhodnout, která ze dvou metod je lepší, je však nemožné.

Podívejme se nyní v krátkosti na výsledky pro kód s objekty ***DataTable* a *DataAdapter*.**

Ve stručnosti lze říci, že pro něj bezezbytku platí to samé, co bylo v předchozím textu napsáno o kódu s objektem *DataReader* – jednoznačně nejpomalejší psqlODBC, lehce zaostávající PGNP a víceméně vyrovnaná rychlost Npgsql a PgSql. Snad jediným překvapujícím výsledkem je rozdíl mezi prvním a dalším vykonáním kódu u PGNP v bodě 4 (vytvoření objektu *OleDbDataAdapter*), jak jej zobrazuje Obrázek A-12.

Poslední dva grafy na Obrázek A-13 a Obrázek A-14 ukazují výsledky pro měření provedená pro **celý blok operací**.

Je zde opět vidět vysoká časová náročnost ODBC, která psqlODBC opět staví „mimo hru“ při porovnávání metod. Z Obrázek A-14 pak lze vyčíst několik zajímavých věcí. Lze říci, že většina času potřebného na vykonání celého bloku kódu padne na kompilaci kódu, což ve výsledku způsobuje, že všechna první spuštění trvají řádově stejnou dobu. Lze také říci, že není výrazný rozdíl v tom, zda data načítáme pomocí objektu *DataReader* či *DataAdapter* a *DataTable*. Tento rozdíl je pouze v případě psqlODBC – a to značný.

Nejdůležitější zjištění nám přináší porovnání naměřených časových údajů poskytovatelů dat Npgsql a PgSql. PgSql vykazuje lepší hodnoty při prvním vykonání kódu. Důležitější však je informace o stráveném čase při dalších vykonáních kódu, která jsou odstíněna od potřeby kompilace kódu, a tedy vystihují čas skutečně potřebný pro komunikaci s databázovým serverem. A zde v obou případech (*DataReader*; *DataAdapter* + *DataTable*) jednoznačně vítězí **Npgsql**.

Pro zajímavost byl proveden také časový test pro **ADO** používající jako OLE DB ovladač PGNP. Z naměřených hodnot vyplývá vyšší časová náročnost při prvním vykonání kódu, kdy je potřeba načíst potřebné součásti pro spuštění neřízeného

kódu ADO. Při dalším spuštění pak – logicky – vykazuje téměř totožné hodnoty jako PGNP s poskytovatelem dat OLE DB.

2.5 Výběr vhodné technologie

Jak již bylo uvedeno, primárním rozhodovacím kritériem při volbě poskytovatele dat pro ADO.NET by mělo být, zda pro daný datový zdroj existuje nativní poskytovatel v řízeném kódu. Tuto podmínku splňují PgSql a Npgsql. I když byly testovány také zbývající dvě technologie psqlODBC a PGNP, výsledky jejich testů nepřinesly nic, co by zdůvodňovalo uvedenou podmínku porušit.

Ze dvou zůstávajících možností se kloní prvenství spíše k Npgsql, i když mnohdy byly výsledky Npgsql a PgSql velmi podobné. Co ale přihrává vítězství zcela na stranu Npgsql je fakt, že se jedná o projekt s otevřeným zdrojovým kódem, což přináší mnohem větší možnosti, než komerční řešení PgSql.

Na základě výsledků této kapitoly bude tedy pro ostatní úkoly této práce použit .NET poskytovatel dat **Npgsql – .Net Data Provider for PostgreSQL**.

3 ORM v .NET Frameworku

Jedním z dílčích úkolů této práce je vytvořit generátor, který by mapoval jednotlivé tabulky v databázi na objekty v jazyce C#. V podstatě se nechá říci, že výsledkem má být jednoduchý objektově-relační (OR) mapper.

Cílem této kapitoly – dříve, než bude popsáno, jak byl daný úkol řešen – je shrnout stávající možnosti objektově relačního mapování (ORM) v .NET Frameworku a to především s pohledu možností využít jej pro přístup k databázím PostgreSQL.

3.1 Silně typový DataSet

Silně typový (strongly-typed) dataset sice není ORM nástroj v pravém slova smyslu, ale své místo zde má, poněvadž umožňuje požadovaný úkol – mapování databázových objektů do objektů CLR.

C# (a celý .NET) je typový jazyk, kde každý objekt musí mít jasně určený typ – na rozdíl od netypových jazyků jako PHP či JavaScript. Stejně tak jsou typové i sloupce v databázích. Obecný dataset, tak jak byl popsán v kapitole 1.7.4, nemá ponětí o tom, jakého typu jsou data, která jsou do něj z databáze načítána, a proto pracuje s obecným typem `Object` a poli tabulek, sloupců a řádků. Takovýto dataset nazýváme jako netypový.

Typový dataset je odvozený z `System.Data.DataSet` a pro deklaraci používá XML Schema Definition (XSD). Ze souboru se schématem (typicky `.xsd`) je vygenerován soubor v jazyce C#, který obsahuje příslušné třídy a objekty typového datasetu. Vygenerování obstará automaticky Visual Studio, případně lze použít utilitu `xsd.exe` z Microsoft Windows SDK (Software Development Kit).

Výhodou typového datasetu je, že je efektivnější než netypový dataset. Díky definovanému schématu není při jeho plnění daty potřeba typová identifikace a konverze, protože typy jsou určeny již při kompilaci. Další z výhod je možnost využití *intellisense*, tedy automatického doplňování kódu ve vývojovém prostředí.

Typový dataset lze využít i pro přístup k databázi PostgreSQL. S poskytovatelem dat Npgsql můžeme získat XSD soubor zavoláním metody `WriteXmlSchema()`.

```
...
NpgsqlDataAdapter da =
    new NpgsqlDataAdapter("select * from shop.\"Products\"", conn);
DataSet ds = new DataSet();
da.Fill(ds);
ds.WriteXmlSchema("ShopProducts.xsd");
...
```

Díky použití typového datasetu se značně zjednodušuje zápis pro přístup k jeho jednotlivým objektů. Namísto klasického zápisu pro netypový dataset

```
string pName = n.Tables["Products"].Rows[0]["product_name"].ToString();
```

lze použít

```
string pName = ds.Product[0].product_name;
```

3.2 LINQ, LINQ to ..., DbLinq

.Net Framework 3.5 a třetí verze jazyka C# přinesly kromě jiných novinek také **Language-Integrated Query (LINQ)**. Ten umožňuje psát dotazy nad různými datovými zdroji pomocí jednotné syntaxe. Pro svůj zápis používá klíčová slova podobná příkazům v jazyce SQL.

```
string[] pozdravy =
    { "Dobrý den", "Hello", "Buenos días", "Guten Tag ", "Bonjour" };
var dlouhePozdravy = from p in pozdravy
                     where p.Length > 5
                     orderby p.Length
                     select p;
```

Z uvedeného příkladu je bez hlubších znalostí na první pohled jasné, že po vykonání tohoto kódu bude v proměnné `dlouhePozdravy` seznam pozdravů delších něž 5 znaků (tedy všechny kromě „Hello“) seřazených podle délky.

Z hlediska jazyka C# se jedná pouze o tzv. syntaktický cukr, kdy jsou jednotlivá klíčová slova přeložena do odpovídajících metod (`Select()`, `Where()`,...). Tyto metody lze rovněž přímo používat, a tak lze předchozí příklad přepsat jako:

```
var dlouhePozdravy = pozdravy.Where(p => p.Length > 5)
                             .OrderBy(p => p.Length)
                             .Select(p => p);
```

Poznámka, že zatím co v předchozím příkladě je zápis `select p` ze syntaktických důvodů povinný, v tomto případě je volání metody `Select()` zbytečné, protože vrátí ten samý výsledek, jako je na jejím vstupu.

Jak lze vidět, základ LINQ tvoří lambda funkce a anonymní delegáty. Všechny metody LINQ lze najít v assembly `System.Core.dll`, konkrétně ve statické třídě `System.Linq.Enumerable`, která definuje extension metody pro datový typ `IEnumerable<T>`. Právě implementace tohoto rozhraní umožňuje nad určitým datovým typem používat LINQ.

3.2.1 LINQ to ...

LINQ, tak jak byl popsán na předchozích řádcích, se někdy nazývá **LINQ to Objects**, a to z toho důvodu, že se používá pro dotazování nad objekty, které jsou v paměti

programu. Dotazovat se lze ale i jiných datových zdrojů a to díky statické třídě `System.Linq.Queryable`, které obsahuje extension metody pro typ `IQueryable<T>`, který rozšiřuje možnosti `IEnumerable<T>`. Zásadní rozdíl je v tom, že `IQueryable<T>` obsahuje vlastnost `Provider` a že jeho metody povětšinou nepřijímají lambda funkci ale *expression tree*, který je předáván ke zpracování právě providerovi (`IQueryProvider`). Ten je pak zodpovědný za zpracování zadaného lambda výrazu, sestavení požadavku a komunikaci s datovým zdrojem.

Přímo v .NET Frameworku je zahrnuta podpora pro dotazování do XML (**LINQ to XML**, `System.Xml.Linq`) či Entity Frameworku (**LINQ to Entities**), o kterém bude zmínka později. Zajímavým řešením je **LINQ to DataSet**, který – jak název napovídá – umožňuje dotazování nad datasetem (typovým i netypovým) a získat z něj typové objekty. Této možnosti bylo několikrát s výhodou použito i při vlastní realizaci této práce v kódu generátoru C# tříd:

```
...
da.Fill(dt); // naplnění objektu typu DataTable z DataAdapteru
var columns = dt.AsEnumerable().Select(
      column => new
      {
            Name = column.Field<string>("column_name"),
            DataType = column.Field<string>("data_type"),
            IsNullable = column.Field<bool>("is_nullable")
      });
```

Posledním a asi i nejznámějším zástupcem rodiny LINQ to … z dílny Microsoftu je **LINQ to SQL**. Pod tímto označením se skrývá jednak provider, umoňující rozhraní `IQueryable<T>` dotazovat se do relační databáze a zároveň **ORM framework**, který tuto databázi mapuje na objekty prostředí CLR. Informace o mapování se uchovávají v souboru `.dbml` a podstatnou vlastností LINQ to SQL je, že toto mapování je 1:1 – tedy jedné tabulce v databázi odpovídá jedna třída v jazyce C#. Kromě těchto tříd obsahuje LINQ to SQL *datový model* ještě třídu označovanou jako datový kontext (*DataContext*), která je zodpovědná za veškeré dění v modelu – zpřístupňuje třídy jednotlivých databázových tabulek, udržuje spojení na databázi, registruje veškeré změny v datech a ty je schopna odeslat ve správném formátu zpět na databázový server a další.

Co je na LINQ to SQL limitující, je skutečnost, že tento provider je napsán tak, aby poskytoval co nejlepší výsledky pro komunikaci s databázovým serverem firmy Microsoft, a proto jej nelze použít pro jiné databáze než Microsoft SQL Server verze 2000 a vyšší.

3.2.2 DbLinq

Nemožnost použít LINQ to SQL pro jinou databázi než MS SQL Server vedla ke vzniku různých projektů, které se snaží poskytovat providery pro další databáze.

Jedním z takových to projektů je **DbLinq** [28], který umožňuje používat LINQ pro databáze MySQL, Oracle, *PostgreSQL*, SQLite, Ingres, Firebird a SQL Server poskytuje API blízké LINQ to SQL. Projekt se od září 2008 nachází ve verzi 0.18 označené jako CTP1 a přes to, že od té doby nebyla uvolněna verze další, nezdá se, že by se jednalo o mrtvý projekt.

Pro přístup k databázi PostgreSQL používá jako poskytovatele dat Npgsql, i když neumí využít všechny jeho vlastnosti. Především si pak neumí při generování DBML schématu a C# tříd datového modelu poradit se speciálními datovými typy PostgreSQL a se sloupci typu pole, pro které generuje chybné výsledky. I přes tento nedostatek je ale DbLinq při použití databázových typů, které mají odpovídající ekvivalenty v CLR, plně použitelným projektem pro přístup k PostgreSQL pomocí LINQ.

3.3 Entity Framework

LINQ to SQL přinesl do .NET Frameworku jednoduchý a výkonný ORM framework, ale pouze pro SQL Server. Pomyslný dluh vůči vývojářům používajících jiné databáze byl napraven vydáním .NET 3.5 SP1 v srpnu 2008, v rámci kterého byl uvolněn **ADO.NET Entity Framework (EF)**.

Jedná se o pokročilý ORM framework, který na rozdíl od LINQ to SQL není spjat s konkrétním poskytovatelem a umožňuje tak použití pro takřka pro libovolnou databázi (za předpokladu, že existuje vhodný .NET poskytovatel dat).

Základní a podstatnou vlastností Entity Frameworku je, že abstrahuje vývojáře od relačního (logického) schématu, v kterém jsou data uložena (např. relační tabulky a sloupce nebo XML tagy), a prezentuje schéma konceptuální, které přestavuje sémantické vztahy mezi daty.

Této abstrakce je dosaženo třemi modely:

1. Model úložiště (SSDL) – popisuje schéma úložiště (většinou tedy databáze), jeho tabulky, sloupce a jejich typ a vztahy mezi nimi.
2. Konceptuální model (CSDL) – poskytuje business pohled na databázi a určuje objektový model. Aplikace ví pouze o tomto modelu.
3. Popis mapování (MDL) – mapuje objekty CSDL na objekty SSDL.

Spojení všech tří modelů se označuje jako **Entity Data Model** (EDM). Modely mohou existovat jako tři samostatné soubory, častěji jsou ale spojovány do jednoho souboru s příponou `.edmx`, pro který má Visual Studio podporu grafického designeru.

Ve výchozím stavu jsou objekty konceptuálního modelu a modelu úložiště mapovány ve vztahu 1:1, což ale lze editací CSDL a MDL modelů upravit na základě požadavků business pohledu do jiného formátu. Například zatím co v relační

databázi musí být vazba M:N rozložena pomocí vazební tabulky na M:1 a 1:N, v EDM lze namodelovat přímo vazbu M:N.

3.3.1 Entity Framework a PostgreSQL

ADO.NET Entity Framework lze použít i pro přístup k databázi PostgreSQL a to díky .NET poskytovateli dat Npgsql, který podporu pro EF zahrnul do své druhé verze, která byla uvolněna jen necelé dva měsíce po oficiálním vydání Entity Frameworku.

Použití Npgsql 2 jako poskytovatele dat pro EF vyžaduje několik podmínek. Předně je třeba z několika různých binárních verzí Npgsql nabízených ke stažení vybrat verzi označenou jako *ms.net3.5*. Následně je třeba získané DLL soubory `Npgsql.dll` a `Mono.Security.dll` (knihovna z projektu Mono, na které má Npgsql závislost) pomocí příkazu `gacutil -i <název souboru>` přidat do *Global Cache Assembly* (GAC). Posledním krokem je modifikace souboru s nastaveními .NET Frameworku `machine.config` a přidání odkazu na *ProviderFactory* poskytovatele Npgsql. To se provede přidáním následujícího kódu do `<system.data>` `<DbProviderFactories>` ve zmíněném souboru.

```
<add name="Npgsql Data Provider" invariant="Npgsql"
     description=".Net Framework Data Provider for PostgreSQL Server"
     type="Npgsql.NpgsqlFactory, Npgsql, Version=2.0.4.0, Culture=neutral,
          PublicKeyToken=5d8b90d52f46fda7"/>
```

Tímto je systém připraven pro použití Npgsql jako poskytovatele dat pro Entity Framework.

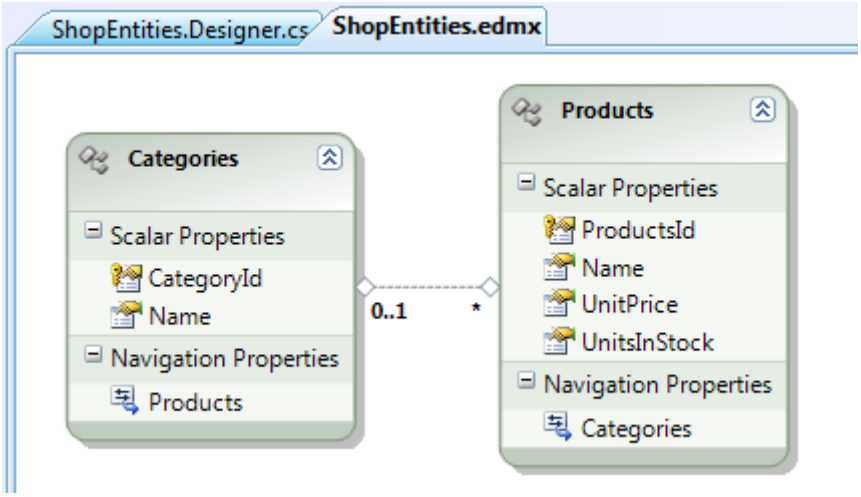
Pro vlastní přístup ke konkrétní databázi pomocí EF je potřeba nejdříve získat EDM model. Ten lze teoreticky vytvořit ručně, což je ale řešení nepraktické, zdlouhavé a především náchylné k chybám. V praxi se proto používá speciálních generátorů. Npgsql zatím nepodporuje integraci do Visual Studia, proto je potřeba vygenerovovat model pomocí příkazové řádky a utility `EdmGen.exe`, která je součástí standartní distribuce .NET 3.5 SP1. Ta se na základě zadaných parametrů připojí k dané databázi, získá potřebné údaje a vytvoří SSDL, CSDL a MDL modely a soubor objektů v jazyce C#. EdmGen nepodporuje přímou tvorbu `.edmx` souboru, který je praktičtější pro použití ve Visual Studiu. Můžeme jej ale získat jednoduchým ručním složením tří vygenerovaných souborů do jednoho, a nebo využít volně dostupný projekt **EdmGen2** [29], který přímé generování `.edmx` souborů umožňuje.

Použití Entity Frameworku pro PostgreSQL má ale i některá omezení, daná vlastnostmi EF. Entity Framework musí být v rámci EDM modelu schopen namapovat sloupce logického modelu do konkrétních datových typů prostředí CLR na straně modelu konceptuálního. Množina datových typů CLR je ale omezená, tvoří ji především `System.String`, `System.Boolean`, `System.DataTime`,

`System.Int32`, `System.Double`, `System.Decimal` a některé další typy. Tyto jsou univerzální napříč různými platformami a lze je použít pro všechny databáze. To ale neplatí o speciálních datových typech databáze PostgreSQL jako je například `point`, `polygon` `circle`, `inet` či `interval`. Proto jsou sloupce těchto typů při generování modelu vynechány a do logického, natož pak konceptuálního modelu, se nezahrnují. Podle slov jednoho z vývojářů Npgsql se do budoucna jeví jako nejpravděpodobnější řešení mapování těchto speciálních typů pro potřeby EF na typ `string`.[30]

3.3.2 Ukázka použití Entity Frameworku

Uvažme logický model databázové struktury PostgreSQL databáze tak, jak je uveden v příloze B Jeho namapováním získáme konceptuální model v podobě, kterou zobrazuje Obrázek 3-1.



Obrázek 3-1: Diagram konceptuálního modelu EF ve Visual Studiu 2008

Následující kód pak ukazuje jednoduché použití tohoto modelu pro získání záznamů z databáze.

```
// získá objektový kontext EF
ShopEntities shop = new ShopEntities();

// vybere všechny produkty s cenou menší než 100, kterých je
// na skladě více jak 10 a seřadí je podle ceny
var products = shop.Products
               .Where(p => p.UnitPrice < 100.0m && p.UnitsInStock > 10)
               .OrderBy(p => p.UnitPrice);

// vypíše do konzole názvy produktů, jejich cenu a počet kusů
foreach (var p in products)
{
  Console.WriteLine("{0}\t{1}\t{2}", p.Name, p.UnitPrice, p.UnitsInStock);
}
```

Jak lze v ukázce vidět, díky LINQ to Entities lze nad *EntitySetem* `Products` provádět LINQ operace. Jejich parametry typu *expression tree* jsou zpracovány providerem `IQueryProvider` a získaný *command tree* je předán *EntityClient data provideru*. Ten dále zavolá odpovídajícího *.NET data providera* (v našem případě Npgsql) a opět mu předá zadaný command tree (tedy stále ne SQL). Teprve .NET poskytovatel dat sestaví ze zadaného stromu příkazů patřičný SQL dotaz v syntaxi odpovídající konkrétní databázi a tím se pak na databázi dotáže. Tímto několika vrstvým modelem je zajištěna univerzálnost Entity Frameworku pro různé databáze. Použití EF pro konkrétní databázi je tedy v podstatě limitováno pouze existencí vhodného .NET poskytovatele dat, který umí zpracovat command tree.

3.4 Další ORM nástroje

Existují i další ORM frameworky, které umožňují mapování objektů databáze PostgreSQL do objektů prostředí CLR, ale na jejichž popis zde není místo. Z těch nejznámějších jmenujme alespoň **NHibernate**, což je port oblíbeného ORM frameworku z prostředí jazyka Java do prostředí .NET. Poznamenejme, že NHibernate pro vlastní přístup k PostgreSQL databázi rovněž používá Npgsql.

Z komerčních produktů pak podporuje ORM například v kapitole 2.1.5 popisovaný **dotConnect for PostgreSQL**, který poskytuje vlastní verzi **LINQ to PostgreSQL** včetně integrace do Visual Studia.

4 NpgObjects - mapování DB tabulek do objektů

Cílem této kapitoly je popsat, jakým způsobem bylo řešeno a dosaženo mapování tabulek databáze PostgreSQL do objektů prostředí CLR, konkrétně pak do tříd v jazyce C#.

Celkově lze problém rozložit do dvou dílčích částí; první je popis struktury samotných C# tříd, které jsou výstupem generátoru a reflektují, mapují strukturu databáze, a druhá je popis objektů a tříd – můžeme tedy říci frameworku – které stojí v pozadí a umožňují funkčnost celého OR mapování.

Pro vlastní řešení bylo zvoleno označení **NpgObjects,** které v sobě odráží skutečnost, že se jedná o řešení pro .NET (N), že se jedná o řešení pro PostgreSQL (pg) a konečně, že se jedná o mapování na objekty (Objects). Toto označení nese jak hlavní třída projektu tvořící jakýsi datový kontext, tak projekt samotný a také jmenný prostor (`MMST.NpgObjects`), ve kterém se všechny potřebné třídy nacházejí.

4.1 Vývojové prostředí

Ještě než se budeme věnovat vlastnímu popisu, poznamenejme, v jakém prostředí bylo celé řešení vyvíjeno a testováno.

Bylo použito vývojové prostředí *Microsoft Visual Studio 2008 SP1* na operačním systému *Windows 7 Beta (build 7000)*, kde byl nainstalován *.NET Framework 3.5 SP1*. Pro vlastní vývoj byl použit programovací jazyk *C# 3*. Jako databázový server bylo použito *PostgreSQL 8.3.6*. a jako .NET poskytovatel dat byl na základě výsledků z kapitoly 2**Chyba! Nenalezen zdroj odkazů.** vybrán *Npgsql – .Net Data Provider for PostgreSQL verze 2.0.4*.

4.2 Generované třídy a systém mapování

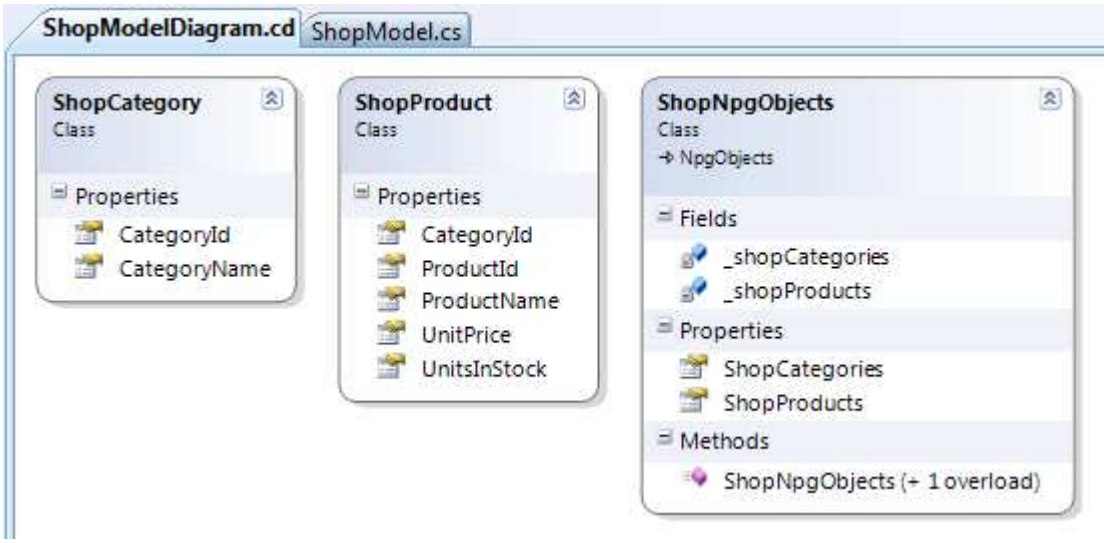
První věc, kterou si je třeba představit pro pochopení toho, jak celý projekt NpgObjects funguje, je struktura C# tříd tvořících základ modelu.

V příloze C je uveden výpis vygenerovaného zdrojového kódu vzorového modelu, který bude sloužit jako příklad, na který bude v následujících odstavcích odkazováno. Uvedený model pak odpovídá databázové struktuře tak, jak je popsána v příloze B .

Pokud si databázovou strukturu lehce rozebereme, jedná se o dvě tabulky pro produkty a jejich kategorie. Každá má definován primární klíč typu `serial`, což je speciální označení pro `integer`, kterému je přiřazena sekvence; jinými slovy, jeho hodnota se automaticky zvyšuje (obdoba volby `AUTO_INCREMENT` u MySQL či `IDENTITY` u databáze SQL Server). Sloupec `category_id` u tabulky produktů slouží jako cizí klíč do tabulky kategorií. Jako jediný ze všech sloupců může nabývat hodnoty `NULL`, což znamená, že přiřazení produktu do kategorie je nepovinné. Pro

obě tabulky bylo vytvořeno zvláštní schéma (tedy něco jako jmenný prostor) s názvem `shop`. Pro jejich adresaci tedy nestačí použít pouhý název tabulky, ale je potřeba přiřadit i název schématu: `shop."Categories"` a `shop."Products"` (poznámka k uvozovkám viz kapitola 1.3.1).

Výsledek namapování databázové strutury do tříd v jazyce C# v podobě diagramů těchto tříd pak zobrazuje Obrázek 4-1.



Obrázek 4-1: Diagram NpgObjects modelu ve Visual Studiu 2008

Při pohledu do kódu v příloze B , respektive na diagram Obrázek 4-1, vidíme, že jednotlivým tabulkám odpovídají třídy `ShopCategory` a `ShopProducts`. Sloupce tabulek jsou pak reprezentovány jako vlastnosti těchto tříd. Třída `ShopNpgObjects` představuje hlavní třídu celého modelu, která při vlastním použití v aplikaci zpřístupňuje ostatní třídy reprezentující tabulky.

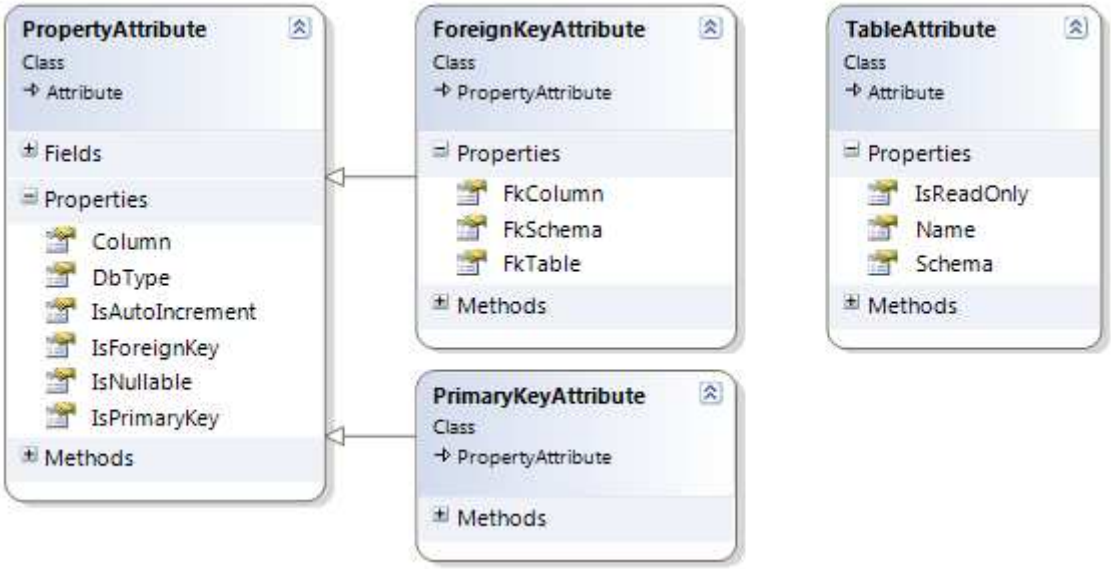
4.2.1 Popis modelu pomocí atributů

Základem mapování tabulek, sloupců a řádků relační databáze do objektů CLR je vytvoření vhodného modelu, který by udržoval informace o struktuře databáze a zároveň poskytoval objektovou strukturu pro použití v aplikacích.

Všechny v předchozí kapitole zmíněné ORM frameworky jako základ pro uchování informací o mapování používají nějaký druh XML dokumentu; ať už je to EDMX u Entity Frameworku, DBML u LINQ to SQL či XSD u typového DataSetu. Pro NpgObjects bylo zvoleno řešení jiné. Ačkoliv se při vlastním generování C# tříd XML také využívá, do konkrétní aplikace si nesou třídy informaci o svoji vazbě na databázi „sami v sobě“ – pracuje se tedy s čistým C# kódem. Konkrétně je pak využito té vlastnosti jazyka C#, že každé třídě, vlastnosti, metodě, … mohou být přiřazeny tzv. atributy.

V projektu NpgObjects jsou definovány celkem čtyři třídy (`TableAttribute`, `PropertyAttribute`, `PrimaryKeyAttribute` a `ForeignKeyAttribute`) odvozené od

`System.Attribute`, jak je ukazuje diagram na obrázku Obrázek 4-2. Pomocí těchto tříd je pak dosaženo popisu vazby jednotlivých prvků NpgObjects modelu na databázi.



Obrázek 4-2: Diagram tříd atributů pro popis NpgObjects modelu

Atribut Table

Atribut `Table` slouží pro označení jednotlivých tříd reprezentujících tabulky. Jeho konstruktor bere jako parametr název tabulky v databázi, přetížená varianta s dvěma parametry umožňuje zadat také název schématu. Pokud tento zadán není, uvažuje se základní schéma public. Parametr `IsReadOnly` umožňuje nastavením na hodnotu *true* označit, že daná třída představuje ve skutečnosti pohled, který pro nás ale nepředstavuje nic jiného než tabulku, do které nelze vkládat záznamy, lze z ní pouze číst.

```
[Table("Products", "shop")]
```

Atribut Property

Atributem `Property` jsou označovány vlastnosti, které přestavují jednotlivé sloupce tabulek. Tento atribut umožňuje zadat celkem čtyři údaje. Předně jsou to název a datový typ sloupce v databázi. Datový typ je zadávám pomocí výčtového typu `NpgsqlDbType` o bude o něm zmínka ještě později. Další dva parametry `IsNullable` a `IsAutoIncrement` umožňují nastavit příznaky, zda je či není daný sloupec v databázi označen jako `NOT NULL` a zda má přiřazenu sekvenci pro automatické zvyšování své hodnoty u nových záznamů. Poslední dvě vlastnosti třídy `PropertyAttribute` `IsPrimaryKey` a `IsForeignKey` jsou pouze pro čtení a v případě atributu Property jsou vnitřně nastaveny na hodnotu *false*. Svoje uplatnění najdou až v kombinaci s dalšími dvěma třídami, které jsou z třídy *PropertyAttribute* odvozeny.

```
[Property(Column = "product_name", DbType = NpgsqlDbType.Varchar,
IsNullable = false)]
```

Atribut PrimaryKey a primární klíče

První z nich je třída *PrimaryKeyAttribute*, která slouží k označení těch vlastností, jejichž sloupce mají v databázi nastaveno omezení (constraint) `PRIMARY KEY`. Atribut `PrimaryKey` svůj rodičovský atribut nikterak nerozšiřuje, pouze v rámci svého konstruktoru nastavuje vlastnost `IsPrimaryKey` na hodnotu *true*.

```
[PrimaryKey(Column = "product_id", DbType = NpgsqlDbType.Integer,
IsAutoIncrement = true)]
```

NpgObjects je navržen tak, aby si poradil s primárním klíčem definovaným nad jedním i nad více sloupci. Zároveň také umí pracovat s tabulkami (mezi které patří i všechny pohledy), které primární klíč definován nemají. Z takovýchto tabulek lze bez problémů číst, lze do nich zapisovat a lze z nich mazat. Co u takovýchto tabulek umožněno není a pokus o to skončí vyvoláním výjimky, je operace `UPDATE`. Poznamenejme však, že definice primárních (a cizích) klíčů je základem správného návrhu databázového modelu a že u dobře navržené databáze by k takové situaci nemělo nikdy dojít.

Atribut ForeignKey a cizí klíče

Druhou třídou odvozenou od atributu `Property` je třída *ForeignKeyAttribute*, která analogicky k předchozímu atributu nastavuje na *true* příznak `IsForeignKey`. Krom toho její konstruktor přijímá tři řetězce pro zadání názvu tabulky, sloupce a schématu pro označení cílového sloupce, na který hodnota cizího klíče ukazuje. Definována je i varianta konstruktoru, který bere pouze dva parametry. V tom případě se zadává pouze tabulka a sloupec a podobně jako u atributu `Table` se automaticky uvažuje základní schéma *public*.

```
[ForeignKey("Categories", "category_id", "shop", Column = "category_id",
DbType = NpgsqlDbType.Integer, IsNullable = true)]
```

Informace o tom, kam daný cizí klíč ukazuje, není v rámci projektu NpgObjects nijak dál využívána. Prosto byla ve výstupu generátoru ponechána, protože si lze představit její využití například při rozšíření třídy, minimálně má však alespoň svoji informační hodnotu při pohledu do zdrojového kódu.

Závěrem popisu atributů sloužících pro mapování vazby mezi databází a jejím objektovým modelem poznamenejme, že tři různé třídy pro popis typu sloupce – ač by se možná mohly zdát, především u primárního klíče, zbytečné – slouží především k vyšší vizuální přehlednosti vygenerovaného kódu, kdy je na první pohled vidět, co je primární klíč, co klíč cizí a co obyčejný sloupec. Zároveň využití dědičnosti a definice všech základních vlastností v rodičovské třídě umožňuje jednoduché procházení všech atributů bez ohledu na typ při jejich zpracování.

4.2.2 Mapování datových typů

PostgreSQL obsahuje bohatou škálu datových typů, mnohem větší než třeba MS SQL Server. Pokud chceme tyto datové typy použít v prostředí .NET, musíme pro ně najít vhodný ekvivalent mezi datovými typy, které .NET (resp. CLR) nabízí. To však nemusí být dostatečné řešení, jak již ostatně bylo zmíněno v kapitole 3.3.1 v souvislosti s generováním modelu pro Entity Framework.

Pokud si s datovým portfoliem CLR nevystačíme, jsme nuceni pro dané databázové typy napsat své vlastní struktury, na které je budeme do prostředí .NET převádět. Tento problém za nás naštěstí řeší samotné Npgsql, které nejrozšířenější speciální datové typy PostgreSQL převádí do takovýchto struktur v rámci Npgsql definovaných.

Tím, že NpgObjects je řešení šité na míru přímo PostgreSQL potažmo Npgsql a není na rozdíl od zmíněného EF vázáno nějakou univerzálností, umožňuje využít naplno veškeré možnosti, které Npgsql pro práci s datovými typy databáze PostgreSQL nabízí.

Je několik základních věcí, které musíme brát v potaz při převodu databázového typu na typ prostředí CLR. Jednak je to samotný datový typ, dále to, zda daný sloupec, o jehož typ se jedná, může v databázi nabývat hodnoty *NULL* a zda zvolený ekvivalent na straně .NETu může tuto hodnotu obsahovat. V neposlední řadě sem vstupuje také skutečnost, že všechny datové typy PostgreSQL mohou být použity jako pole.

Nullable datové typy

Datové typy v prostředí CLR dělíme do dvou základních typů – referenční a hodnotové (viz 1.5.1 Common Type System (CTS)). Zatímco referenční typy, mezi které patří především typ String, mohou bez problémů nabývat hodnoty *NULL*, u typů hodnotových, kam patří všechny číselné typy, DateTime a obecně všechny struktury, je to jiné. Jak jejich název napovídá, musejí obsahovat nějakou hodnotu – a tou hodnota *NULL* z tohoto pohledu není.

Jak se tedy postavit k problému, že některé hodnotové typy mohou na straně databáze nabývat hodnoty *NULL*? Řešení je jednoduché a nabízí jej sám .NET. V druhé verzi přibyly do jazyka C# tzv. nullable typy. Ve skutečnosti se jedná o jednu generickou třídu `System.Nullable<T>`, kterou lze použít na všechny struktury. V praxi se pak většinou používá zkrácený zápis pomocí otazníku. Například datový typ `int?` není nic jiného než zkratka pro `System.Nullable<int>`. A jak vidno, toto už není hodnotový typ, ale typ referenční, který může hodnotu *NULL* bez problémů obsahovat.

Vztahy mezi datovými typy v databázi a prostředí CLR

Tabulka Tab. E-1 v příloze E ukazuje vztah mezi datovými typy na straně databáze a datovými typy použitelnými v prostředí .NET. Zároveň také ukazuje, jaký typ je

potřeba použít, aby mohl obsahovat hodnotu *NULL*. Poznamenejme, že některé uvedené databázové typy mají více aliasů, a tak např. `int4` může být označen jako `integer` a `varchar` jako `character varying`.

Ve zmíněné tabulce jsou také uvedeny hodnoty výčtového typu *NpgsqlDbType*, které jsou přiřazovány vlastnosti `DbType` u atributů `Property`, `PrimaryKey` a `ForeignKey`, jak již bylo uvedeno dříve. Jejich uvedení v rámci atributů je nutné pro správnou funkci Npgsql při komunikaci s databází.

Datový typ pole

Typ *NpgsqlDbType* obsahuje speciální hodnotu `NpgsqlDbType.Array`, která slouží k označení, že daný typ je pole. Konkrétní typ pole se pak určí jako bitový OR hodnoty `Array` s hodnoty požadovaného typu. Budeme-li tedy uvažovat např. že daný sloupec obsahuje pole hodnot typu integer (v databázi označeno jako `_int4`), pak u zmíněných atributů bude vlastnosti `DbType` přiřazena hodnota následujícím způsobem:

```
DbType = NpgsqlDbType.Integer | NpgsqlDbType.Array;
```

V souvislosti s datovým typem polem uveďme ještě krátkou poznámku o jeho vztahu k nullable datovým typům. Mohlo by se zdát matoucí a chybné, že pro sloupce typu pole použijeme v objektovém modelu na straně .NETu stejný typ, ať je sloupec označený jako `NOT NULL` nebo není. Zůstaneme-li u příkladu s typem integer, v obou případech bude vlastnost modelu mapující daný sloupec deklarována jako `int[]`, i když bychom v případě nullable sloupce možná očekávali typ `int?[]`.

Zde je třeba si uvědomit, že typ `int[]` není typem hodnotovým, ale že se jedná o typ referenční odvozený z abstraktní třídy *System.Array*, který sám o sobě může obsahovat hodnotu *NULL*. Typ `int?[]` tedy není primárně typem, který by mohl obsahovat hodnotu *NULL*, ale typem, který obsahuje hodnoty, které mohou být null, jak ukazuje následující příklad.

```
int[] pole1 = { 1, 2, 3 };      // int[] obsahuje pole hodnot typu int
int[] pole2 = null;             // int[] může být null
int?[] pole3 = null;            // int?[] může být také null...
int?[] pole4 = { 1, null, 3 }; // a null může být i některá z jeho hodnot
```

4.2.3 Datový kontext modelu

Třída ***NpgObjects*** (v našem vzorovém modelu *ShopNpgObjects*) představuje jakýsi datový kontext, podobně jako je tomu u Entity Frameworku či LINQ to SQL. Na rozdíl od těchto má ale mnohem menší zodpovědnost.

Jeho nejdůležitější vlastností je zpřístupňování jednotlivých objektů přestavujících databázové tabulky. Činí tak díky svým vlastnostem typu `NpgTable<T>`, kde typ `T`

představuje jednotlivé typy všech tříd v modelu. V našem případě je to tedy `NpgTable<ShopProduct>` a `NpgTable<ShopCategory>`.

Konkrétní objekty daného typu jsou získávány voláním generické metody `GetTable<T>()` z rodičovské třídy *NpgObjects*. Tato metoda se ale volá pouze při prvním požadavku na danou tabulku a získaný objekt se ukládá do privátní proměnné shodného typu a při dalších požadavcích na „tabulku" je vracen stále ten samý objekt. Tím je dosaženo toho, že v rámci jednoho objektu *NpgObjects, tedy v rámci jednoho kontextu, existuje vždy jen jedna instance konkrétního typu.

```
public NpgTable<ShopProduct> ShopProducts
{
    get
    {
        if (this._shopProducts == null)
            this._shopProducts = base.GetTable<ShopProduct>();
        return this._shopProducts;
    }
}
private NpgTable<ShopProduct> _shopProducts;
```

Druhou vlastností třídy *NpgObjects je, že ve svém konstruktoru přijímá buď *connection string* nebo přímo objekt typu NpgsqlConnection. O toto spojení s databází se pak na úrovni své rodičovské rodičovské stará a poskytuje jej dalším objektům v NpgObjects projektu.

4.2.4 Částečné třídy modelu

Mezi další vlastnosti, kterých si můžeme na vygenerovaném modelu všimnout, patří to, že všechny třídy jsou označeny klíčovým slovem `partial`, což je umožňuje jednoduše a přehledně rozšiřovat o další metody ve zvláštním souboru odděleném od vlastního modelu.

Pokud budeme uvažovat, že se náš vzorový model nachází v souboru `ShopModel.cs`, pak můžeme vytvořit například soubor `ShopModel.partial.cs` s následujícím kódem.

```
public partial class ShopProduct
{
    public decimal GetCenaSDani()
    {
        return this.UnitPrice * 1.19m;
    }
}
```

Tímto jednoduchým způsobem můžeme nad vygenerovaným modelem jednoduše definovat rozšiřující metody, které nám usnadní práci s modelem. Konkrétně

uvedený příklad nám přímo na objektech typu Product umožňuje volat metodu `GetCenaSDani()` a tím jednoduše získat cenu s daní bez nutnosti nějakého externího dopočítávání. (Ponechme stranou správnost či nesprávnost konstanty 1,19 přímo ve zdojovém kódu, příklad nám slouží pouze jako demonstrační.)

4.2.5 Omezení NpgObjects modelu

Při návrhu modelu byly schválně zahrnuty některé zjednodušující předpoklady. Porovnáme-li zatím popsaný model NpgObjects s v kapitole 3 popisovanými modely Entity Frameworku a LINQ to SQL, dojdeme k závěru, že NpgObjects se blíží spíše LINQ to SQL. Neexistuje zde rozdělení na třívrstvý model mapování s logickým a konceptuálním modelem spojenými vrstvou mapovací, jako je tomu u EF. Naopak shodně s LINQ to SQL jsou databázové tabulky mapovány na objekty v jazyce C# přímo 1:1, tedy jedné tabulce odpovídá jedna třída, jednomu sloupci této tabulky odpovídá jedna vlastnosti této třídy.

NpgObjects zachází ve zjednodušování poněkud dále, kdy každou tabulku–třídu považuje za samostatný objekt bez vazby na další objekty. Zatím co u LINQ to SQL lze tabulky v dotazech spojovat (JOIN) nebo se přímo z jednoho objektu odkazovat na vlastnosti objektu určeného cizím klíčem, v NpgObjects toto **nelze**.

```
// tohoto s NpgObjects nedosáhneme
var categoryName = concreteProduct.Categories.CategoryName;
```

Potřebujeme-li takovouto funkcionalitu, můžeme ji suplovat pomocí pohledů na straně databáze a jim odpovídajících objektů na straně C# modelu. NpgObjects je určeno pro jednodušší projekty bez složité databázové struktury, kdy je toto chování považováno za dostačující.

Takovéto chování používá i příklad ***ExampleWebShop*** na přiloženém DVD, který data pro svoji úvodní stránku načítá právě z pohledu.

4.2.6 Jmenné konvence modelu

Poslední věc, kterou zmiňme v souvislosti s modelem NpgObjects, jsou použité jmenné konvence. Jak bude ještě uvedeno v souvislosti s generátorem, název jmenného prostoru a hlavní třídy (datového kontextu) modelu lze explicitně zadat při generování modelu. Pokud tak není učiněno, jsou jejich názvy automaticky určeny jako `<NázevDatabáze>Model` a `<NázevDatabáze>NpgObjects`.

Názvy tříd a vlastností jsou pak generovány zcela automaticky a to podle následujících pravidel. Jako základ se používá pochopitelně název tabulky či sloupce, který, pokud je to možné, je převed to tzv. *CamelCase* (resp. *PascalCase*) syntaxe. V případě, že se tabulka nachází v jiném schématu, než je schéma public, je na začátek názvu odpovídající třídy přidán název tohoto schématu. To platí i pro náš vzorový model, kdy se tabulky *Products* a *Categories* nacházejí ve schéma `shop` a proto jsou názvy tříd v modelu pojmenovány jako `ShopProduct` a `ShopCategory`.

Tímto je zajištěno, že nedojde ke kolizi v případě, že různá schémata v rámci jedné databáze obsahují tabulky stejných názvů.

Druhé věci, které si je možné všimnout u názvů tříd, je převedení názvu z množného čísla na jednotné. Toto chování vychází z hojně rozšířeného principu pojmenovávání, kdy se pro názvy databázových tabulek obsahujících množství záznamů používá množné číslo, kdežto pro třídu, jejíž instance reprezentuje pouze jediný objekt (jeden řádek tabulky), se používá čísla jednotného. Naproti tomu – ve shodě s touto logikou – jsou názvy vlastností hlavní třídy *NpgObjects představující celé tabulky uváděny v čísle množném.

Generátor modelu tedy rozlišuje, zda je název v množném čísle, a případně jej převede na jednotné číslo. Toto vzhledem ke složitosti češtiny funguje z pochopitelných důvodů pouze pro anglické názvy tabulek. Pokud generátor nazná, že převod nelze uskutečnit, je všude použit původní název tabulky.

4.3 Mapovací framework

V předchozím textu je popsán způsob mapování databázových tabulek do tříd v jazyce C# a dále jakými prostředky je toto mapování v modelu uloženo. Samotný model nám ale nestačí, potřebujeme něco, co jej bude „oživovat“. Tím je sada C# tříd, které můžeme nazvat jako mapovací framework.

Základní třídy tohoto frameworku jsou vytvořeny jako generické. To umožňuje jejich přímé použití nad konkrétním typem-třídou reprezentujícím databázovou tabulku bez nutnosti mít tyto třídy jako abstraktní. Tím je značně redukováno množství kódu, které je pro každý model nutné generovat a které by v případě abstraktních tříd bylo mnohem větší.

4.3.1 Třída NpgObjects<T>

O této třídě již byla řeč v souvislosti datovým kontextem modelu, že zpřístupňuje objekty NpgTable<T> a spravuje připojení k databázi. K informacím uvedeným v kapitole 4.2.3 doplňme, že třída NpgObjects<T> implementuje rozhraní *IDiposable*, což umožňuje instanci této třídy (resp. instanci potomka této třídy) použít společně s klíčovým slovem using, o kterém byla zmínka v kapitole 2.4.3.

4.3.2 Třída NpgTable<T>

Třída NpgTable<T> představuje asi nejdůležitější třídu z celého projektu. Reprezentuje jednotlivé tabulky a nad nimi umožňuje provádět operace INSERT, UPDATE a DELETE a přijímá parametry pro operaci SELECT.

Její konstruktor je označen modifikátorem přístupu internal, což zajišťuje, že se instance této třídy bude vyskytovat pouze v rámci datového kontextu třídy NpgObjects<T>, protože takto označené členy tříd jsou dostupné pouze v rámci jedné assembly (tedy DLL NpgObjects) a nelze tedy jinde objekt tohoto typu vytvořit.

Vkládání, editace a mazání záznamů

Vkládání, editaci a mazání záznamů můžeme vysvětlit společně, protože z pohledu třídy NpgTable<T> mezi nimi není téměř žádný rozdíl.

NpgTable<T> má tři veřejné metody `Insert(T item)`, `Update(T item)`, `Delete(T item)`, které přijímají za svůj parametr objekt typu T, kde T zastupuje typ třídy reprezentující databázovou tabulku. Tento parametr je při do seznamu `IList<T> _listForInsert`, `IList<T> _listForInsert`, `IList<T> _listForInsert` podle typu metody.

Pro všechny operace existují také veřejné metody `Clear*()` (znak * znamená Insert/Update/Delete), které umožňují ten který konkrétní seznam předčasně vyčistit.

K řádnému vyčištění všech seznamů dojde po úspěšném vykonání veřejné metody `SaveChanges()`, která za pomoci neveřejných metod `get*Query()` a `execute*Query()` provede nad danou databázovou tabulkou požadované změny. Tato operace se celá provádí v rámci transakce a to v pořadí INSERT, UPDATE, DELETE, aby byla zajištěna konzistence dat v případě chyb.

Několik poznámek k sestavování dotazů metodami `get*Query()`. Při operaci INSERT se objekt vkládá do databáze tak, jak je zadán; jediné, co se bere v potaz, je, zda některý sloupec je označen jako *IsAutoIncrement*. V tomto případě se daný sloupec (ve většině případů se bude jednat o primární klíč) z příkazu INSERT vynechá, a to i v případě, že některý objekt má ve své vlastnosti tomuto sloupci odpovídající nějakou hodnotu přiřazenu. Tímto je zabezpečeno, že se do tabulky v databázi automaticky doplní správná hodnota daná odpovídající sekvencí.

Jak již bylo uvedeno dříve, příkaz UPDATE nelze provést nad tabulkou bez přiřazeného primárního klíče. Pokud je tato podmínka splněna, je omezující příkaz WHERE sestaven ze všech sloupců tvořících primární klíč.

Při sestavování příkazu pro DELETE je rovněž klausule WHERE sestavena z primárních klíčů. Pokud tabulka primární klíč nemá, použijí se pro výběr mazaných řádků všechny sloupce.

Metody `execute*Query()` postupně procházejí odpovídající seznamy `_listFor*<T>` a pomocí Npgsql vykonávají příkazy sestavené metodami `get*Query()`. Hodnoty jsou do příkazů zadávány pomocí parametrů (*NpgsqlParameter*), čímž by měla být zajištěna bezpečnost a odolnost vůči útoku *SQL injection*.

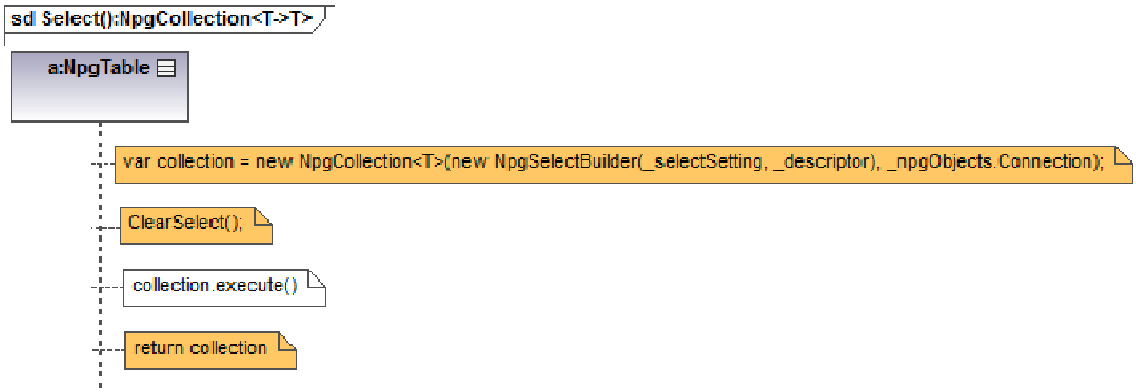
Získávání záznamů z databáze

Třída NpgTable<T> není za vlastní získávání záznamů z databáze zodpovědná. Pouze poskytuje metody, pomocí kterých se nechají zadat podmínky a parametry určující požadované záznamy.

Třída NpgTable<T> obsahuje privátní proměnnou `_selectSetting` typu *NpgSelectSetting*. Tato třída má definovány tři vlastnosti *Where*, *Limit* a *Order*, které umožňují přijímat nastavení pro sestavení příkazu SELECT. Z toho vyplývá, že proměnná `_selectSetting` představuje jakýsi kontejner na zadané parametry, které se do něj ukládají v okamžiku zavolání některé z metod třídy NpgTable<T> `Where()`, `Order()` či `Limit()`.

Select()

Podíváme-li se do zdrojového kódu metody Select(), s překvapením zjistíme, že má pouhé čtyři řádky. Jak již bylo uvedeno, NpgTable<T> za vykonání příkazu SELECT není zodpovědný a toto minimum kódu stačí na to, aby byla předána zodpovědnost za výběr dat z databáze objektu typu NpgCollection<T>. Tento typ je také návratovým typem metody Select() a bude o něm řeč za chvíli.



Obrázek 4-3: Sekvenční diagram metody NpgTable<T>.Select()

Jak lze vidět z uvedeného sekvenčního diagramu na obrázku Obrázek 4-3, metoda Select() vytváří novou instanci třídy NpgCollection<T>. Té předává objekt Connection pro připojení k databázi a v rámci konstruktoru další třídy *NpgSelectBuilder* objekt `_selectSetting` s požadovanými parametry na SELECT.

Toto nastavení je následně vyčištěno, aby byl objekt NpgTable<T> připraven na sestavování nového dotazu. Důležité je volání metody execute() na objektu `collection`, která provede vlastní získání záznamů z databáze. Vytvořený objekt `collection` se získanými daty je nakonec vrácen jako výstup metody.

SelectFirst()

Tato metoda najde uplatnění v okamžiku, kdy očekáváme od databáze pouze jediný výsledek. Je v podstatě pouze obalem nad metodou Select(), kdy ze získané kolekce, která může teoreticky obsahovat více záznamů, vrátí první záznam.

Prepare()
Metoda Prepare() je velice podobná metodě Select(). Liší se v tom, že nevolá na vytvořeném objektu NpgCollection<T> metodu execute() a tím nedochází k okamžitému načítání dat z databáze.

Tato vlastnost najde uplatnění ve dvou případech. Za prvé, pokud z nějakého důvodu chceme oddálit vykonání dotazu do okamžiku, kdy už je jasné, že data jsou skutečně potřeba. Druhá možnost je v případě, že chceme zjistit pouze počet položek odpovídajících zadanému dotazu. Potom je zbytečné vyčítat všechna data z databáze, ale stačí se zavoláním metody Count() zeptat databáze pouze přímo na počet záznamů.

Select(int pageSize)
Tato metoda je opět podobná metodě Select() bez parametrů. Rozdíl je v tom, že místo objektu NpgCollection<T> vrací NpgPagedCollection<T>. Svoje uplatnění najde především v kombinaci s komponentou PagedDataGridView popisovanou v kapitole 6.

4.3.3 Třídy NpgCollection<T> a NpgPagedCollection<T>
Hned v úvodu pro předejití nedorozumění poznamenejme, že ačkoliv třídy NpgCollection<T> a NpgPagedCollection<T> obsahují ve svém názvu slovo Collection, tak neimplementují rozhraní *ICollection<T>*. Jak už bylo možné pochopit z předchozího textu, tyto „kolekce“ obsahují data získaná z databáze po zavolání metody Select() (nebo některé jí podobné metody) a tato data jsou v rámci této kolekce určena pouze ke čtení (pro editaci a další máme metody Insert(), Update(), Delete). Bylo by tedy nežádoucí implementovat rozhraní *ICollection<T>*, které obsahuje právě i metody pro přidávání, odstraňování, … záznamů.

Třída NpgCollection<T> se pro uživatele na vnější pohled chová, jako by se jednalo například o List<T> nebo podobný typ naplněný daty získanými z databáze. Ve skutečnosti se ale uvnitř této třídy skrývá celá logika pro dotazování se na databázi, získávání záznamů a jejich prezentaci jako odpovídajících objektů.

execute()
Jádro této třídy tvoří metoda execute(), jejíž zjednodušený sekvenční diagram ukazuje Obrázek 4-4. V okamžiku požadavku na získání dat vytvoří objekt příkazů NpgsqlCommand, kterému předá SQL dotaz sestavený třídou NpgSelectBuilder a otevře spojení na databázi. Prostupným procházením výsledku metody ExecuteReader() jsou vytvářeny nové objekty typu T (tedy třídy, která danou databázovou tabulku mapuje), plněny získanými daty a ukládány do seznamu. Pro vytváření a bindování nových objektů se používá následujících metod třídy Type:

```
// vytvoření objektu typu T
var obj = (T)typeof(T).GetConstructor(...);
// nabinduje v objektu obj vlastnost s názvem n hodnotou v objektu o
typeof(T).InvokeMember("n", ... | BindingFlags.SetProperty, null, obj, o);
```

Implementace IEnumerable<T>

Třída NpgCollection<T> implementuje rozhraní *IEnumerable<T>*, což přináší minimálně dvě výhody. Jednak lze přes data získaná z databáze jednoduše iterovat. A zadruhé je možné nad získanými daty používat operace LINQ, a tak data dále třídit, řadit a jinak zpracovávat. Pokud nebyla metoda execute() zavolána v metodě Select(), je tak učiněno automaticky právě v okamžiku, kdy se pokusíme získat Enumerator.

Vlastnosti NpgPagedCollection<T>

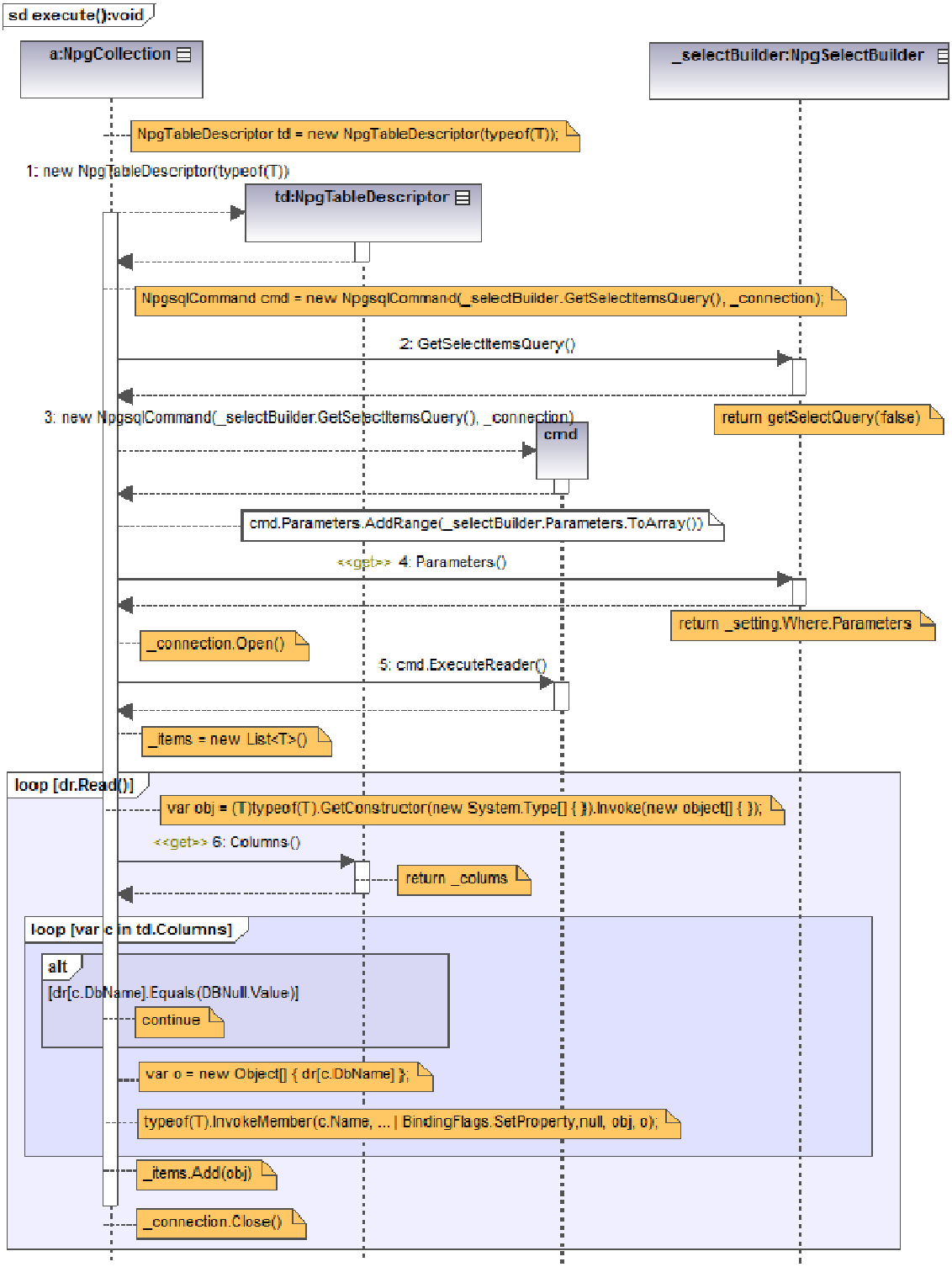
Třída NpgPagedCollection<T> byla vytvořena pro potřeby komponenty PagedDataGridView. Je také asi hlavním důvodem, proč je logika pro dotazování databáze vyčleněna mimo třídu NpgTable<T> a není pohromadě s metodami pro INSERT, UPDATE a DELETE.

Jak název třídy napovídá, slouží ke stránkování dat. Oproti své rodičovské třídě NpgCollection<T> bere ve svém konstruktoru navíc jeden parametr a tím je právě velikost požadované stránky. Tato hodnota je dále předána objektu NpgSelectBuilder, který na jejím základě sestavuje dotazy pro stránkování. Dotaz, který by vracel ohromné množství dat, tak lze rozložit na několikrát a data zpracovat či zobrazit po menších dávkách.

Na úrovni SQL je stránkování řešeno pomocí klíčových slov LIMIT a OFFSET, která příkazu do databáze PostgreSQL umožňují zadat maximální počet vrácených záznamů (LIMIT) a posun prvního vráceného záznamu vůči počátku (OFFSET).

Při testování stránkování nad velkým počtem dat (cca 10^6 záznamů) bylo učiněno zjištění, že čím vyšší offset, tím déle trvá databázovému serveru zpracování dotazu. Byly učiněny nějaké pokusy se stránkováním pomocí serverových kurzorů a příkazů MOVE a FETCH, ale i těmto trvalo zpracování dotazu nad daty „ke konci tabulky“ delší dobu. Ta se subjektivně jevila přibližně stejně velká, jako u použití limitu, a tak byla pro stránkování ponechána kombinace LIMIT – OFFSET, která má snadnější implementaci.

NpgPagedCollection<T> přidává k metodám a vlastnostem NpgCollection<T> některé další. Z vlastností to jsou především PageNumber a PageCount vracející číslo aktuální stránky (číslováno od 1) a celkový počet stránek, z metod pak NextPage() a PreviousPage() umožňující přechod na další/předchozí stránku a Page(int pageNumber) sloužící pro přechod na konkrétní stránku. Všechny tři metody vracejí logickou hodnotu, zda daná stránka existuje (tj. jestli už jsme se nedostali mimo rozsah počtu stránek).



Obrázek 4-4: Zjednodušený sekvenční diagram metody NpgCollection<T>.execute()

Ačkoliv je třída NpgPagedCollection<T> určena především pro spolupráci projektu NpgObjects s komponentou PagedDataGridView, lze ji využít i samostatně a to například v konzolové aplikaci, kde ji lze použít k postupnému výpisu na obrazovku. Něco podobného ukazuje příklad ***ExampleIPagedDataSourceConsole*** na přiloženém DVD.

Poznamenejme, že postupné procházení stránek pomocí konstrukce `while(pagedColl.NextPage())` lze přirovnat k chování objektu typu *DataReader*, když na něm voláme metodu `NextResult()`.

5 Generátor C# tříd NpgObjects modelu

V kapitole 4.2 bylo popsáno, jakým způsobem se v projektu NpgObjects provádí mapování databázových tabulek na objekty prostředí CLR a jak vypadá model v podobě tříd v jazyce C#. Nyní zbývá popsat, jakým způsobem je model generován.

Získání modelu se provádí pomocí generátoru, jehož vlastní realizace byla rozdělena do dvou částí. V první byla vytvořena knihovna, která má vlastní generování modelu na starosti, a následně byla vytvořena aplikace tuto knihovnu využívající.

5.1 Knihovna NpgObjectsGenLibrary

Knihovna NpgObjectsGenLibrary obsahuje dvě hlavní a několik pomocných tříd, které plní tři hlavní úkoly této knihovny – získání popisu struktury databáze, vlastní vytvoření modelu a mapování datových typů.

5.1.1 Třída NpgObjectsSchema

Třída NpgObjectsSchema slouží k získávání informací z databázového serveru a své služby poskytuje jak třídě NpgObjectsGenerator, tak vnější aplikaci, která tuto knihovnu používá. Pomineme-li některé přetížené a odvozené metody, tak tato třída obsahuje tři metody, které nás zajímají. Jsou to GetDatabases(), GetTables() a GetColumn().

GetDatabases()

Jak název napovídá, úkolem této metody je vrátit seznam všech databází na databázovém serveru, ke kterému jsme připojeni. Přetížená varianta s parametrem booleovského typu umožňuje zadáním false vrátit skutečně všechny databáze na serveru. Do výpisu se tak dostanou i jinak skryté systémové databáze s názvy template.

GetTables(string database, TableType tableType)

Funkce GetTables() bere dva parametry. Prvním je název databáze, z které chceme touto funkcí získat seznam tabulek, druhým pak hodnota výčtového typu TableType určující, zda chceme získat informace o skutečných tabulkách, pohledech nebo všech dohromady.

Údaje jsou z databáze získávány z tzv. informačního schématu (*information schema*), což je napříč databázovými systémy standardizované schéma (*information_schema*) obsahující pohledy na systémová data. Z těchto pohledů lze tedy získat metadata o tabulkách, sloupcích a dalších objektech databáze. Většinu požadovaných informací o tabulkách se nám tak podaří získat z jediného pohledu `information_schema.tables`.

Krátká poznámka k sloupci `full_name`. Jeho hodnota je skládána z názvu tabulky a schématu za pomoci regulárních výrazů a slouží pro nás v rámci generátoru jako jedinečný identifikátor tabulky v databázi. S přihlédnutím k pravidlům o uvozování názvů objektů v databázi PostgreSQL (1.3.1) nabývá sloupec `full_name` hodnot v závislosti na názvu schématu a tabulky podle pravidel zřejmých z tabulky Tabulka 5.1.

Tabulka 5.1: Závislost hodnoty sloupce full_name na názvu schématu a tabulky

table_schema	table_name	full_name
public	nazev_tabulky	nazev_tabulky
public	Název tabulky	"Název tabulky"
nazev_schematu	nazev_tabulky	nazev_schematu.nazev_tabulky
nazev_schematu	Název tabulky	nazev_schematu."Název tabulky"
Název schématu	Název tabulky	"Název schématu"."Název tabulky"

Stejná pravidla pro sestavení celého názvu tabulky se používají i v projektu NpgObjects.

GetColumns(string database, string table, string schema)

Metoda GetColums() vrací dle očekávání informace o sloupcích v zadané databázi, schématu a tabulce. Ze všech tří dotazů (databáze vs. tabulky vs. sloupce) je tento nejsložitější a vzniká spojením celkem pěti pohledů informačního schématu (jsou to pohledy `key_column_usage`, `table_constraints`, `referential_constraints`, `constraint_catalog` a `columns`). Takto složitý dotaz je nutné sestavit pro získání informací o primárních a cizích klíčích.

5.1.2 NpgObjectsGenerator

Třída NpgObjectsGenerator je ta, s kterou pracujeme, pokud chceme získat model mapování databázových tabulek do C# tříd. Pro svoji funkci vystavuje řadu vlastností, jejichž nastavením můžeme ovlivňovat výslednou podobu modelu.

Parametry generátoru

Konstruktor přebírá čtyři základní údaje pro připojení k databázi – název serveru, číslo portu a uživatelské jméno a heslo. Dále pak můžeme nastavovat tyto vlastnosti:

Database
Určuje databázi, jejíž model se bude generovat.

Tables a Views
Jedná se o dva seznamy typu IList<string>, které přijímají názvy tabulek/pohledů, které se mají zahrnout do generovaného modelu. Pod pojmem název se rozumí řetězec `full_name`, tak jak jej vrací metoda `NpgObjectsSchema.GetTables()`.

Pokud je hodnota dané vlastnosti *null*, tedy neobsahuje ani prázdný seznam, považuje se to za znamení, že se mají do modelu zahrnout všechny nalezené tabulky/pohledy. Toto řešení značně ulehčuje práci při použití knihovny např. v konzolové aplikaci, kde by bylo obtížné zadávat všechny názvy tabulek.

ModelName
Zadaný řetězec představuje název pro hlavní třídu modelu.

ModelNamespace
Určuje jmenný prostor modelu.

SaveFilePath
Určuje cestu a název souboru, kde se má vygenerovaný model uložit.

SaveXml
Nastavením na true se spolu s vygenerovaným modelem uloží do zvláštního souboru XML dokument, který byl ke generování modelu použit.

Generování modelu

K vygenerování modelu dojde po zavolání jediné veřejné metody Generate().

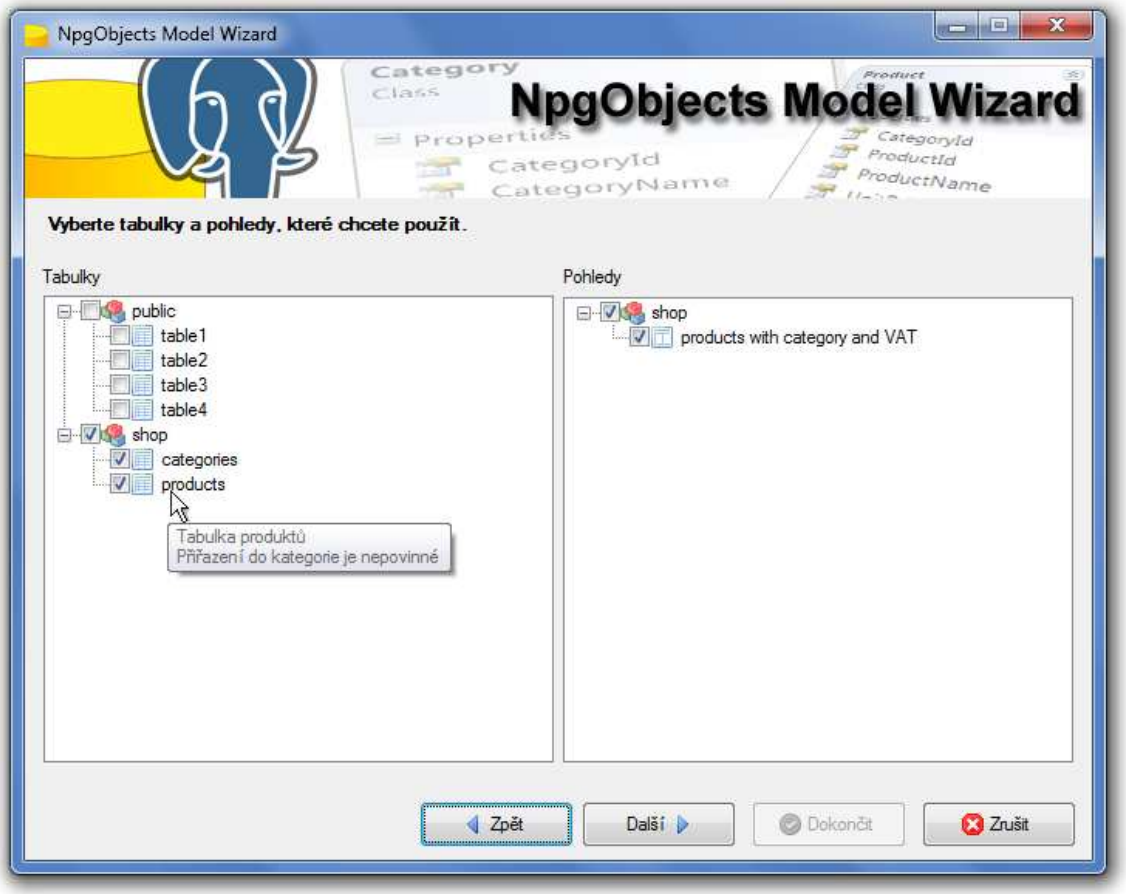
Na základě hodnot vlastností `Tables` a `Views` se určí, které tabulky/pohledy se mají do modelu zahrnout. Pomocí metod třídy *NpgObjectsSchema* jsou o nic a jejich sloupcích získány z databáze potřebné informace, které jsou sestaveny do XML dokumentu. Výsledný XML dokument pro vzorovou databázi z přílohy B ukazuje výpis v příloze D

Součástí knihovny NpgObjectsGenLibrary je soubor `generator.xslt`, který obsahuje styly transformací, které se aplikují na sestavený XML dokument. Výsledek transformace se pak uloží jako soubor zadaný ve vlastnosti `SaveFilePath`. Konečný soubor modelu v jazyce C# pro naši vzorovou databázi pak představuje již dříve zmíněný výpis kódu v příloze C

5.2 Aplikace NpgObjectsGenWizard

Aplikace NpgObjectsGenWizard slouží k pohodlnému vygenerování NpgObjects modelu pomocí sady formulářů. Postupně uživatele provede jednotlivými nastaveními a volbami.

Nejzajímavější obrazovkou je výběr tabulek a pohledů, které se mají zahrnout do modelu. Bylo použito komponenty *TreeView*, která umožňuje přehledně zobrazit jednotlivé tabulky a pohledy rozdělené podle schématu, ve kterém se nacházejí. Zároveň, pokud mají schémata, tabulky a pohledy v databázi uveden komentář, je tento při najetí na odpovídající položku zobrazen v podobě *ToolTip* nápovědy.



Obrázek 5-1: Ukázka aplikace NpgObjectsGenWizard

5.3 Aplikace NpgObjectsGen (npgogen.exe)

Jedná se o konzolovou aplikaci využívající knihovnu NpgObjectsGenLibrary, která umožňuje vygenerovat NpgObjects model pomocí příkazové řádky. Na rozdíl od grafického wizardu neumožňuje zadat konkrétní výčet tabulek či pohledů, které se mají zahrnout do modelu. Lze pouze rozhodnout, zda se má model generovat jen pro tabulky nebo jen pohledy nebo oboje.

Utilitu npgogen lze ovládat pomocí následujících parametrů:

Povinné parametry:

```
-d:database       Název databáze jejíž model se bude generovat.
-o:output         Název (cesta) souboru pro uložení modelu.
```

Volitelné parametry:

```
-u:userName       Název uživatele databáze.
-w:server         Heslo uživatele.
-s:server         Název/adresa databázového serveru.
-p:port           Port databázového serveru.
-m:modelName      Název NpgObjects modelu.
-n:namespace      Jmenný prostor modelu.
-t                Generovat pouze model pro tabulky.
-v                Generovat pouze model pro pohledy.
-x                Uložit XML model.
```

6 PagedDataGridView

Dílčím zadáním této práce je navrhnout obecný grid pro zobrazování databázových dat a to tak, aby si poradil s rozsáhlými databázovými tabulkami, které obsahují velké množství dat. Požadavkem – byť nepsaným – je, aby tento grid uměl spolupracovat s navrženým systémem mapování databázových tabulek do objektů prostředí .NET NpgObjects.

6.1 Práce s rozsáhlými daty v .NET

S přihlédnutím k popisu frameworku ADO.NET uvedenému v kapitole 1.7 lze o prostředí .NET říci, že pro práci se zdroji rozsáhlých dat není primárně navrženo. Toto lze tvrdit díky minimálně dvěma věcem. Zaprvé je to absence podpory serverových kurzorů umožňujících postupné načítání a zpracování dat, ale především vlastní podstatou ADO.NET, kdy jeho pomyslné jádro tvoří objekt DataSet, tedy in-memory databáze. Představa, že takto v paměti programu uchováváme off-line kopii databázového úložiště obsahující řádově třeba milióny záznamů, není zrovna ideální.

Ne vždy potřebujeme daná data držet v paměti pro nějaké další zpracování, někdy si vystačíme s jejich pouhým zobrazením. Jak je na tom .NET v tomto případě? Nabízí nějakou možnost efektivního zobrazování velkého počtu dat, nebo nás bude nutit naplnit zobrazovací komponentu DataGridView všemi daty najednou?

6.1.1 Vlastnost VirtualMode komponenty DataGridView

Nutno říci, že na tuto potřebu vývojáři komponenty DataGridView mysleli, a tak tato poskytuje některé své vlastnosti a události, díky kterým lze načítat zobrazovaná data postupně.

Lze tak učinit nastavením vlastnosti VirtualMode na hodnotu true, čím DataGridView říkáme, že jeho interakci s datovým zdrojem budeme řídit my. Abychom tak mohli činit, je potřeba vytvořit obslužnou metodu pro událost CellValueNeeded. Minimálně tato a případně některé další události umožňují řídit tok dat z datového zdroje do komponenty.

Zjednodušeně lze říci, že pokud je posunem v tabulce zobrazena nějaká buňka, je vyvolána událost CellValueNeeded. Ta očekává, že jí obslužná metoda na základě předaného indexu řádku a sloupce do argumentu události nastaví odpovídající hodnotu, která má být v dané buňce zobrazena. V kombinaci s nějakou vhodnou implementací cache lze dosáhnout vcelku efektivního výsledku, kdy má uživatel dojem, že má v tabulce zobrazena všechna data, ale ve skutečnosti se z celého množství dat načítá jen počet řádků daných velikostí cache. Podrobnější informace s ukázkami kódu jsou dostupné na stránkách firmy Microsoft v rámci MSDN Library [31] [32].

6.2 Návrh principu PagedDataGridView

O uvedeném řešení načítání dat pomocí virtuálního módu lze říci, že má jednu nevýhodu. Tím, že poskytuje uživateli iluzi, že jsou v komponentě načtena všechna data, se stává grid hůře ovladatelným a špatně se v něm naviguje – jen malé posunutí scrollbaru může znamenat skok třeba i o tisíce záznamů. Navržené řešení se snaží tento nedostatek odstranit, a tak kromě řízení toku dat mezi datovým zdrojem a komponentou bere v úvahu i uživatelskou přívětivost.

Jako nejvhodnější varianta bylo shledáno tzv. stránkování, při kterém jednoduše splníme oba požadavky. Jednak dělením rozsáhlého datového zdroje na stránky umožňujeme jednoduchou redukci množství načítaných dat. A zadruhé, stránkování přímo vybízí k implementaci navigačních prvků, pomocí kterých lze přesně nastavovat svoji pozici v rozsáhlém množství záznamů.

Zamyslíme-li se nad tím, v kterém místě by se měla stránkovací logika nacházet, dojdeme k závěru, že to není samotná komponenta stránkovacího gridu (nazývejme ji ***PagedDataGridView***), ale objekt – může také říci poskytovatel dat – označovaný jako DataSource, tedy datový zdroj. Takovýmto datovým zdrojem může být buď přímo objekt, který obsahuje nějaká data, nebo objekt-poskytovatel, který zpřístupňuje data v jiném datovém zdroji, typicky databázi.

Toto chování je dáno skutečností, že nám nejde v první řadě o to data stránkovat (to je víceméně vedlejší bonus zvoleného řešení), ale řídit jejich tok. Řízení na straně komponenty by bylo vhodné v případě, že by nám šlo o postupné zobrazování již načtených dat, pokud ale chceme řídit načítání dat, je třeba tuto zodpovědnost „kolik-čeho-odkud“ přenést na datový zdroj.

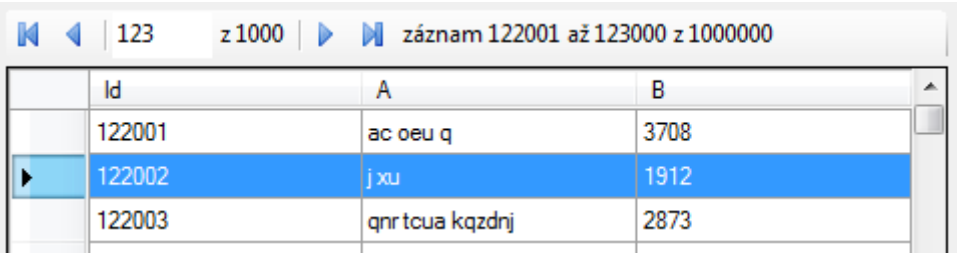
6.2.1 Interface jako univerzální řešení

Jak má tedy ale stránkovací grid poznat, kterou stránku zobrazuje? A jak vůbec pozná, že zadaný datový zdroj je stránkován? K tomuto účelu bylo navrženo speciální rozhraní ***IPagedDataSource***. Každý seznam, kolekce či jiný objekt, který má být pomocí PagedDataGridView stránkován, musí toto rozhraní implementovat. A je už jedno, jestli se jedná o objekt, který má všechna data v paměti počítače a chce je pouze stránkovat, nebo jestli je to objekt, který metody tohoto rozhraní použije k řízení toku dat z databáze či jiného zdroje.

V rámci této práce bylo rozhraní *IPagedDataSource* implementováno třídou *NpgPagedCollection<T>* projektu **NpgObjects**, což umožňuje součinnost tohoto jednoduchého ORM frameworku s navrženou komponentou *PagedDataGridView.* Obecně by ale rozhraní IPagedDataSource mělo být schopné umožnit stránkování jakéhokoliv zdroje dat.

6.3 Komponenta PagedDataGridView

Grid PagedDataGridView byl navržen jako *Windows Forms* komponenta za použití třídy *System.Windows.Forms.UserControl*. Tento uživatelský kontrol pak obsahuje DataGridView pro vlastní zobrazování dat a *ToolStrip* bar pro ovládací prvky navigace.



Obrázek 6-1: Komponenta PagedDataGridView s daty

Jak lze vidět na obrázku Obrázek 6-1, navigace zaujímá místo na vrchu komponenty a grid pro data se rozkládá pod ním. Navigace dále obsahuje čtyři tlačítka „První“, „Předchozí“, „Další“ a „Poslední“ pro přechod na odpovídající stránku a textové pole, které vypisuje číslo aktuální stránky a umožňuje zadat požadovanou stránku. Kromě toho jsou vypisovány také informace o celkovém počtu stránek a všech záznamů.

Mnohé vlastnosti vnořené komponenty DataGridView lze nastavovat pomocí vlastností zmíněných v dalším textu, některé jsou naopak napevno definovány a měnit je nelze. Mezi vlastnosti, které měnit nejdou, patří především vlastnost *ReadOnly* nastavená na *true* a *SelectionMode* nastavená na *FullRowSelect*. První volba zajišťuje, že buňky nelze editovat, protože PagedDataGridView je určen pouze pro výpis dat, druhá volba pak nutí grid označovat jako vybraný celý řádek.

DataGridView na sobě odchytává událost stisku klávesy a tím umožňuje navigaci (stránkování) pomocí klávesnice. Rozpoznává tyto klávesy a jejich kombinace:

- **Šipky doprava a doleva** – přechod na další a předchozí stránku se zachováním čísla aktuálního řádku
- **Šipka dolů na posledním řádku** – přechod na první řádek další stránky.
- **Šipka nahoru na prvním řádku** – přechod na poslední řádek předchozí řádky
- **Home a End** – přechod na první/poslední řádek v rámci aktuální stránky
- **Ctrl + šipka doprava nebo doleva** – přechod na poslední nebo první stránku se zachování čísla aktuálního řádku
- **Ctrl + Home** – přechod na první záznam první stránky
- **Ctrl + End** – přechod na poslední záznam poslední stránky

6.4 Vlastnosti a události PagedDataGridView

Kromě vlastností a událostí pocházejících od rodičovské třídy *UserControl* vystavuje *PagedDataGridView* řadu vlastních vlastností a událostí. Některé jsou vztaženy ke komponentě jako celku, jiné souvisí především s obsaženým gridem. Uveďme zde ze všech alespoň ty, které nejvíce představují a definují chování této stránkovací komponenty.

6.4.1 Vlastnosti

ShowPagesCount a ShowItemsCount
Umožňují zadat, zda se má v navigaci vypisovat počet stránek a informace o celkovém počtu záznamů.

NavigatorPageCountMask a NavigatorItemsCountMask
Určují masku formátu, v jakém se informace o počtu stránek a záznamů vypisují. Výchozí hodnoty jsou `z {0}` a `záznam {0} až {1} z {2}`.

ShowNavigator
Nastavením na *false* možňuje skrytí navigačního panelu a roztažení gridu přes celou plochu komponenty.

Columns
Umožňuje zadat seznam sloupců pro DataGridView.

Poznamenejme, že jelikož je DataGridView začleněn dovnitř uživatelské komponenty, nelze pro návrh sloupců použít designer Visual Studia a je nutné je nadefinovat ručně.

AutoGenerateColumns
Určuje, zda se mají sloupce vygenerovat automaticky ze zadaného zdroje dat.

DefaultCellStyle
Určuje styl a vzhled sloupců.

MultiSelect
Určuje možnost vybrat najednou více řádků tabulky.

DataSource
Umožňuje zadat zdroj dat.

CurrentRowDataBoundItem
Vrací objekt z datového zdroje zadaného do DataSource, který je přiřezen aktuálně vybranému řádku.

AllowUserToDeleteRows
Umožňuje uživateli vyvolat stiskem klávesy Delete událost pro mazání řádku.

6.4.2 Události

ButtonFirstClick, ButtonPreviousClick, ButtonNextClick a ButtonLastClick
Jednotlivé události jsou vyvolány při kliku na odpovídající tlačítko navigace.

PageNumberSubmit
Událost vyvolána stiskem klávesy Enter na textovém poli určujícím číslo stránky, tedy při pokusu načíst stránku požadovaného čísla.

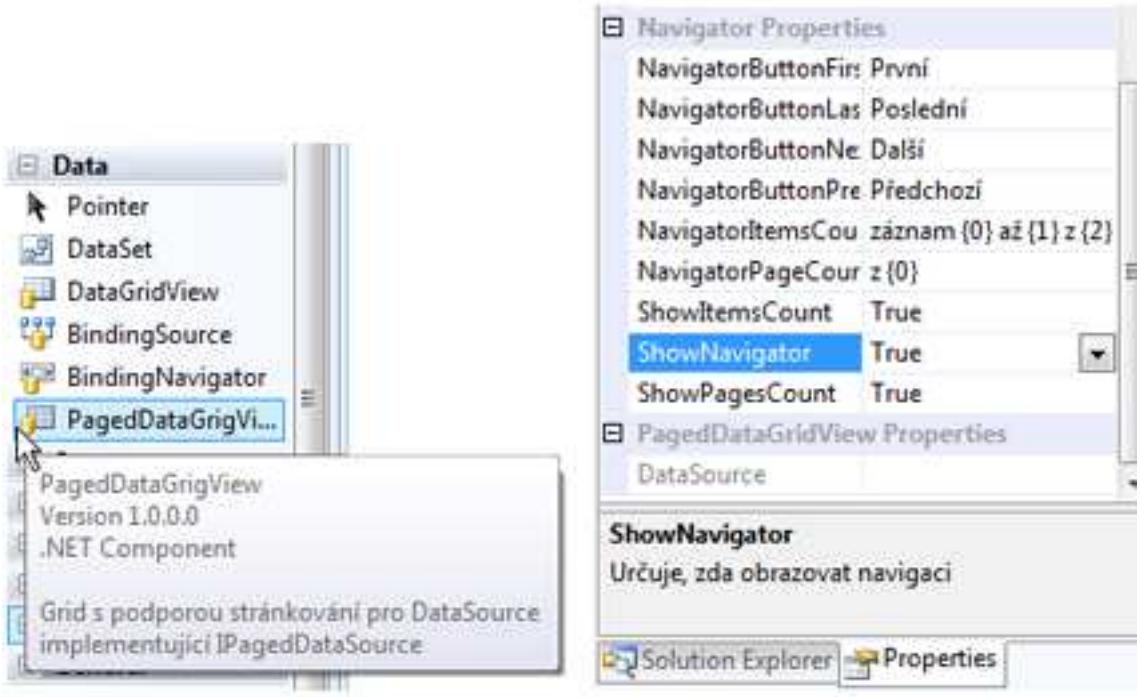
RowEnter a RowLeave
Jedná se o události komponenty DataGridView, které jsou vyvolány při vstupu na řádek a jeho opuštění. Jejich odchycení lze např. použít pro editaci dat.

DeletedRow a DeletingRow
Opět pouze ven z komponenty vystavené události DataGridView pro řízení mazání řádků.

6.5 PagedDataGridView a Visual Studio

Všechny vlastnosti a události komponenty PagedDataGridView jsou ve zdrojovém kódu označeny atributy. Ty je rozdělují do odpovídajících skupin a přiřazují jim popisek a případné další vlastnosti, jako je třeba výchozí hodnota.

Tohoto je využito ve vývojovém prostředí Visual Studio, kde jsou vlastnosti a události přehledně zobrazeny. Integraci komponenty PagedDataGridView do Visual Studia ukazuje obrázek Obrázek 6-2.



Obrázek 6-2: Ukázka integrace komponenty PagedDataGridView do Visual Studia. Integrace do Toolboxu (vlevo) a karty vlastností (vpravo).

7 Závěr

Název této práce *Metody přístupu k databázím PostgreSQL v .NET Framework* pod sebou skrývá ukryto několik dílčích úkolů. Jejich společným jmenovatelem ale stále zůstává databázový systém PostgreSQL, platforma .NET, jazyk C# a práce s databázovými daty.

Toto docela značné rozpětí různých technologií si vyžádalo, aby jejich popisu a představení byla v první části tohoto textu věnována patřičná pozornost. Proto byly postupně představovány vlastnosti a principy databázových systémů obecně, zvláštní část byla věnována podrobnějšímu popisu konkrétního databázového systému PostgreSQL. Dále byly představeny základy platformy .NET, která se poslední dobou dostává stále více do popředí v různých oblastech informačních technologií, a také základy programovacího jazyka C#, který je s touto platformou úzce spjat. A protože vše v této práci se točí okolo práce s daty, bylo nezbytné ozřejmit základy ADO.NET, které na platformě .NET tvoří základ jakékoliv práce s daty a datovými zdroji.

Jelikož .NET neposkytuje přímou oficiální podporu pro spolupráci s databází PostgreSQL, byla ve druhé kapitole tohoto textu zmapována množina technologií, které lze pro přístup k PostgreSQL z .NETu použít. Z této množiny je na základě testů rychlosti jednotlivých technologií a s přihlédnutím k dalším kritériím vybrán nejvhodnější poskytovatel dat. Tím je ze čtyř porovnávaných produktů zvolen *Npgsql - .NET data provider for PostgreSQL*, a to nejen díky své rychlosti. Úspěšným kritériem je také to, že je celý napsán v řízeném kódu a neméně důležitá je jeho otevřenost zdrojového kódu. Tento zvolený poskytovatel dat byl poté úspěšně používán při dalších úkolech.

Jedním z dílčích úkolů této práce bylo vytvoření sady tříd, které by umožňovaly mapování databázových tabulek do tříd v jazyce C#. Tohoto bylo úspěšně dosaženo vytvořením frameworku *NpgObjects*, který zajišťuje jednak vlastní mapování a zároveň poskytuje třídy a metody, pomocí kterých lze nad vygenerovanými objekty provádět operace SELECT, INSERT, UPDATE i DELETE.

Pro uložení modelu popisujícího vazbu jednotlivých tříd a jejich vlastností na konkrétní tabulky a sloupce v databázi byl zvolen způsob, kdy jsou tyto informace ukládány společně s kódem tříd pomocí atributů těchto tříd a jejich vlastností. Zvolené řešení se ukázalo jako dostačující, dokonce jako výhodné. Jednak se není třeba zabývat jiným souborem s popisem modelu. A další výhodou je, že metainformace uložené přehledným způsobem přímo spolu s kódem tříd, zvyšují přehlednost těchto tříd, protože je na první pohled zřejmé, co která třída či její vlastnost reprezentuje.

Jelikož NpgObjects pro svoji funkci používá Npgsql, může těžit z vlastností tohoto poskytovatele dat. Především v tom, že projekt NpgObjects je díky tomuto

poskytovateli schopen pracovat s velkým množstvím speciální datových typů databáze PostgreSQL.

NpgObjects se sice co do robustnosti či množství nabízených funkcí nemůže rovnat velkým projektům, jako je třeba Entity Framework, přesto poskytuje ucelenou množinu funkcí ORM frameworku, které lze dobře použít v menších projektech.

Vlastní NpgObjects model je generována speciálně navrženým generátorem v podobě DLL knihovny. Díky tomuto řešení lze nad knihovnou postavit libovolnou aplikaci pro generování modelu. V rámci této práce byly vytvořeny dvě aplikace – grafický wizard umožňující pohodlné nastavení generovaného modelu a konzolová aplikace pro práci z příkazové řádky.

Posledním problémem, který tato práce řešila, bylo zobrazování dat z rozsáhlých databázových tabulek. Bylo zvoleno řešení, kdy jsou data zobrazována po blocích, tzv. stránkách. Navržená komponenta *PagedDataGridView* pracuje v součinnosti s projektem NpgObjects a to díky speciálně navrženému rozhraní IPagedDataSource. Implementací tohoto rozhraní libovolným datovým zdrojem lze tento grid použít pro stránkování i bez vazby na NpgObjects.

Komponenta PagedDataGridView kromě možnosti řídit tok dat do aplikace poskytuje bohaté uživatelské rozhraní pro pohodlné navigování mezi stránkami.

Všechna vytvořená řešení jsou funkční a plní požadovanou funkci. Přesto především u NpgObjects zůstává spousta věcí, které by bylo možné vylepšovat a rozšiřovat. Mezi jinými by to mohla být například implementace vazeb mezi jednotlivými třídami představující databázové tabulky či zadávání příkazů v metodě Where() a dalších pomocí lambda výrazů, a další.

Bibliografie

1. Oppel, Andrew. *Databáze bez předchozích znalostí.* [překl.] Computer Press. Vydání první. Brno : Computer Press, 2006. str. 240. ISBN 80-251-1199-7.

2. Žák, Karel. Historie relačních databází. *Root.cz (www.root.cz), informace nejen ze světa Linuxu.* [Online] 19. října 2001 . [Citace: 15. prosince 2008.] <http://www.root.cz/clanky/historie-relacnich-databazi/>. ISSN 1212-8309.

3. PostgreSQL: About. *PostgreSQL.* [Online] 2008. [Citace: 15. prosince 2008.] <http://www.postgresql.org/about/>.

4. Slovník - Pgsql.cz. *Pgsql.cz.* [Online] 7. července 2007. [Citace: 15. prosince 2008.] <http://www.pgsql.cz/index.php/Slovník>.

5. PostgreSQL 8.3.5 Documentation. *PostgreSQL.* [Online] PostgreSQL Global Development Group, 2008. <http://www.postgresql.org/docs/8.3/interactive/index.html>.

6. Blum, Richard. *PostgreSQL 8 for Windows.* New York : McGraw-Hill, 2007. ISBN 0071485627.

7. Using ODBC with Microsoft SQL Server. *MSDN: Microsoft Developer Network.* [Online] Microsoft Corporation, září 1997. [Citace: 17. prosince 2008.] <http://msdn.microsoft.com/en-us/library/ms811006.aspx>.

8. ODBC Drivers. *OpenLink Software.* [Online] [Citace: 17. prosince 2008.] <http://uda.openlinksw.com/odbc/>.

9. McClure, Wallace B., a další. *Professional ADO.NET 2 Programming with SQL Server 2005, Oracle®, and MySQL®.* Indianapolis : Wiley Publishing, Inc., 2006. ISBN-13: 978-0-7645-8437-4.

10. ECMA-335. *Common Language Infrastructure (CLI) Partitions I to VI.* Geneva : Ecma International, 2006. 4th Edition.

11. Hanák, Ján. *Přecházíme z jazyka Visual Basic 6.0 na jazyk Visual Basic 2005 - Příručka pro programátory, vývojáře, softwarové odborníky a IT specialisty.* Praha : Microsoft. Dostupné on-line: http://www.microsoft.com/cze/msdn/csebooks/default.mspx.

12. Rubiolo, Daniel, a další. Proč Microsoft .NET? [překl.] Dalibor Kačmář. *Přehled architektury .NET.* Dostupné on-line: http://www.microsoft.com/cze/msdn/csebooks/default.mspx.

13. Hanák, Ján. *Visual Basic .NET 2003 - začínáme programovat.* První vydání. Praha : Grada Publishing, 2004. str. 180. ISBN 80-247-0864-7.

14. Kovacs, James. C#/.NET History Lesson. *James Kovacs' Weblog.* [Online] 7. září 2007. [Citace: 3. prosince 2008.] <http://www.jameskovacs.com/blog/CNETHistoryLesson.aspx>.

15. ECMA-334. *C# Language Specification.* Geneva : Ecma International, 2006. 4th Edition.

16. Brust, Andrew J. a Forte, Stephen. *Mistrovství v programování SQL Serveru 2005.* [překl.] Petr Matějů a Jiří Fadrný. Vydání první. Brno : Computer Press, 2007. str. 848. ISBN 978-80-251-1607-4.

17. Malik, Sahil. *Pro ADO.NET 2.0.* Berkeley : Apress, 2005. ISBN: 1-59059-512-2.

18. Esposito, Dino. *ASP.NET a ADO.NET: tvorba dynamických webových stránek.* [překl.] Jaroslav Černý. Vydání první. Praha : Grada Publishing, 2003. str. 352. ISBN 80-247-0474-9.

19. Juřek, Michael. ADO.NET - ResultSet je mrtev - je důvod truchlit? *BonzBlog Michaela Juřka.* [Online] 24. září 2004. [Citace: 13. listopadu 2008.] <http://blog.vyvojar.cz/mjurek/archive/2004/09/24/1893.aspx>.

20. MacDonald, Matthew a Szpuszta, Mario. *ASP.NET 2.0 a C# - tvorba dynamických stránek profesionálně.* [překl.] Jan Pokorný, Petr Kadlec a Erika Lencová. Vydání první. Brno : Zoner Press, 2006. str. 1376. ISBN 80-86815-38-2.

21. psqlODBC Project Home Page. [Online] <http://psqlodbc.projects.postgresql.org/>.

22. PgFoundry: PostgreSQL OLE DB provider for Windows. *Project Info.* [Online] [Citace: 17. prosince 2008.] <http://pgfoundry.org/projects/oledb/>.

23. *The Oledb-devel Archives .* [Online] <http://pgfoundry.org/pipermail/oledb-devel/>.

24. *Npgsql - .Net Data Provider for Postgresql.* [Online] [Citace: 17. prosinec 2008.] <http://npgsql.projects.postgresql.org/>.

25. *PostgreSQL Native OLEDB Provider (PGNP).* [Online] <http://www.pgoledb.com/>.

26. High Perfomance ADO.NET Provider for PostgreSQL with Significantly Improved Abilitie. *dotConnect for PostgreSQL.* [Online] [Citace: 17. prosince 2008.] <http://www.devart.com/dotconnect/postgresql/>.

27. Sample Databases: Project Info. *PgFoundry.org.* [Online] <http://pgfoundry.org/projects/dbsamples/>.

28. Linq Provider for MySql, Oracle and PostgreSQL. *DbLinq Project.* [Online] 15. dubna 2008. [Citace: 22. května 2009.] <http://code2code.net/DB_Linq/>.

29. EdmGen2.exe - Home. *MSDN Code Gallery.* [Online] Microsoft Corporation. <http://code.msdn.microsoft.com/EdmGen2>.

30. Cooley, Josh. Discussion Forums: open-discussion. *PgFoundry: Npgsql .Net Data Provider for Postgresql.* [Online] 3. února 2009. [Citace: 20. května 2009.] <http://pgfoundry.org/forum/message.php?msg_id=1004548>.

31. Walkthrough: Implementing Virtual Mode in the Windows Forms DataGridView Control. *MSDN Library.* [Online] Microsoft Corporation, 2009. [Citace: 22. května 2009.] <http://msdn.microsoft.com/en-us/library/15a31akc.aspx>.

32. Implementing Virtual Mode with Just-In-Time Data Loading in the Windows Forms DataGridView Control. *MSDN Library.* [Online] Microsoft Corporation, 2009. [Citace: 22. května 2009.] <http://msdn.microsoft.com/en-us/library/ms171624.aspx>.

33. Pizzo, Michael a Cochran, Jeff. OLE DB for the ODBC Programmer. *MSDN: Microsoft Developer Network.* [Online] Microsoft Corporation, březen 1997. [Citace: 17. prosince 2008.] <http://msdn.microsoft.com/en-us/library/ms810892.aspx>.

34. Momjian, Bruce. *PostgreSQL: Praktický průvodce.* [překl.] Jan Gregor. Vydání první. Brno : Computer Press, 2003. str. 424. ISBN 80-722-6954-2.

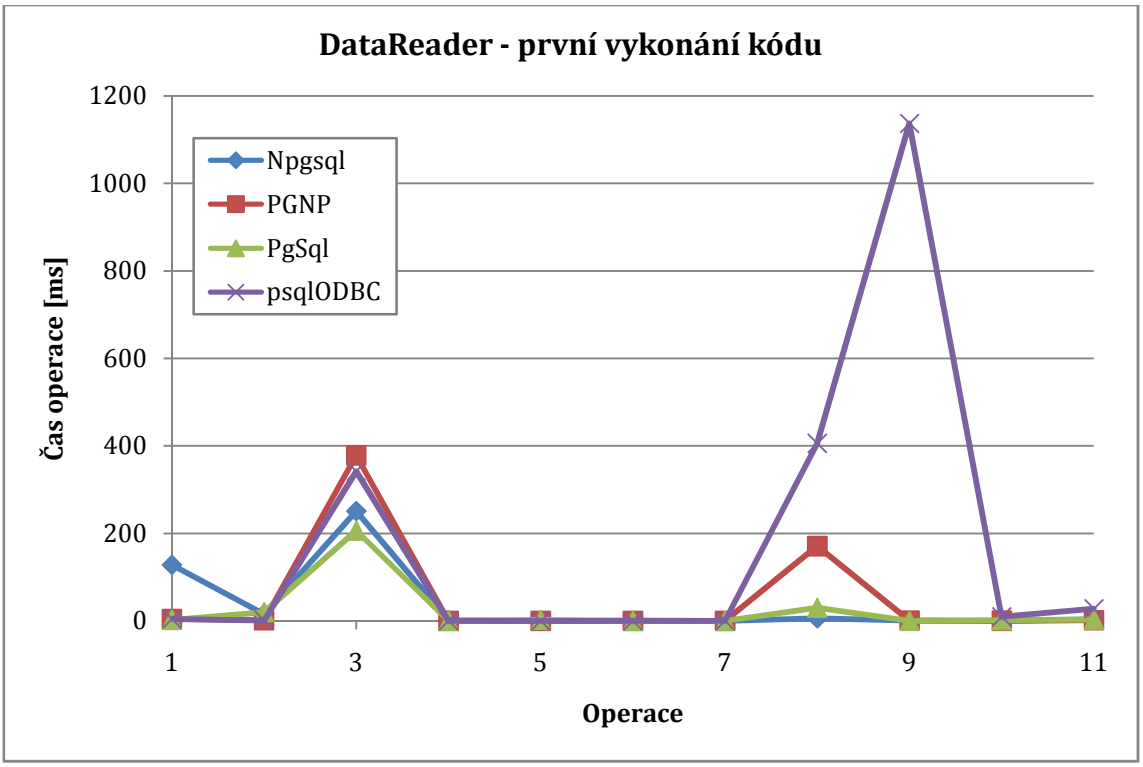
Seznam zkratek

ACID	Atomic, Consistent, Isolated, Durable - atomičnost, konzistence, izolace, trvanlivost
ADO	ActiveX Data Objects
API	Application Programming Interface - rozhraní pro programování aplikací
BCL	Base Class Library
CIL	Commnon Intermediate Language
CLI	Call Level Interface
CLI	Common Language Infrastructure
CLR	Common Language Runtime
CLS	Commnon Language Specification
COM	Component Object Model
CSDL	Conceptual Schema Definition Language
CTS	Commnon Type System
DAO	Data Access Objects
DBMS	Database Management Systems
DCL	Data Control Language - jazyk pro řízení dat
DDL	Data Definition Language - jazyk pro definici dat
DLL	Dynamic Link Library
DML	Data Manipulation Language - jazyk pro manipulaci s daty
DQL	Data Query Language - jazyk pro dotazování
ECPG	Embedded SQL in C
EDM	Entity Data Model
EF	Entity Framework
FCL	Framework Class Library
GAC	Global Assembly Cache
ISM	Information Management System

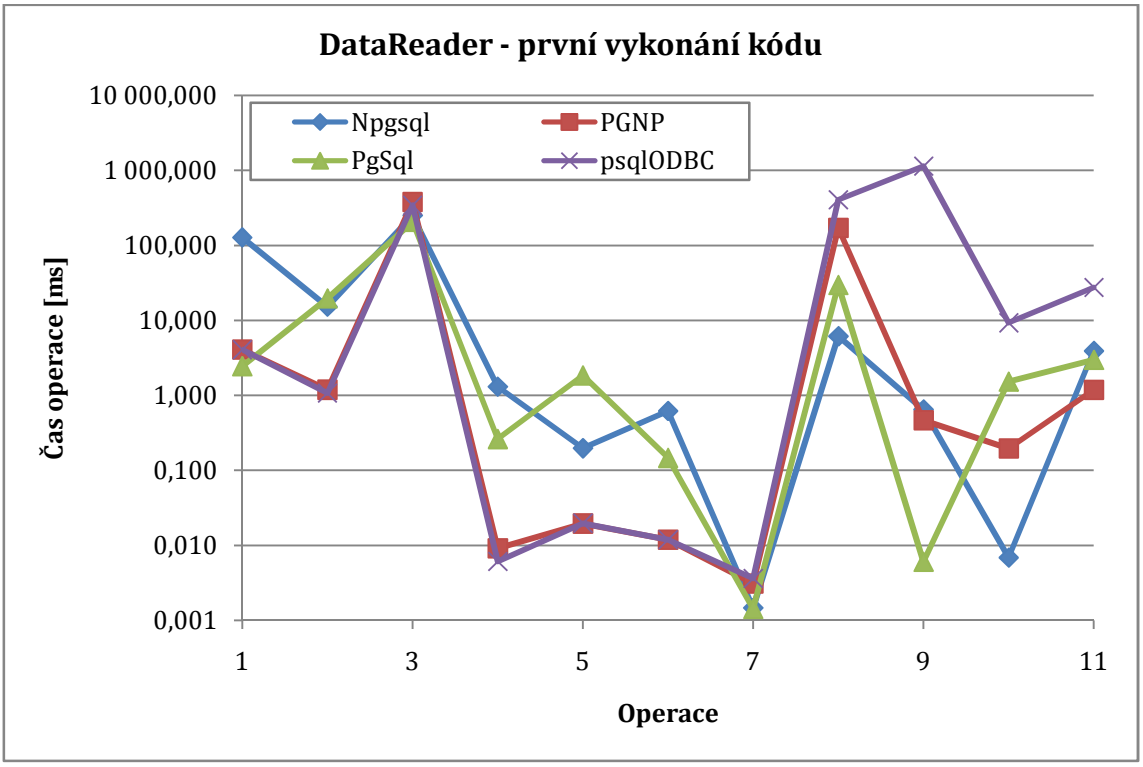
JIT	Just-In-Time
JVM	Java Virtual Machine
LGPL	Lesser General Public License
LINQ	Language Integrated Query
MDAC	Microsoft Data Access Components
MSL	Mapping Specification Language
MSIL	Microsoft Intermediate Language
Npgsql	.NET data provider for PostgreSQL
ODBC	Open Database Connectivity
OLE DB	Object Linking and Embedding Database
OOP	Objektově orientované programování
PGNP	PostgreSQL Native OLEDB Provider
PL/pgSQL	Procedural Language/PostgreSQL Structured Query Language
RDO	Remote Data Objects
SAG	SQL Access Group
SEQUEL	Structured English Query Language
SQL	Structured Query Language - strukturovaný dotazovací jazyk
SŘBD	Systém řízení báze dat
SSDL	Storage Schema Definition Language
VES	Virtual Execution System
XML	eXtensible Markup Language - rozšiřitelný značkovací jazyk
XSD	XML Schema Definition

Seznam příloh

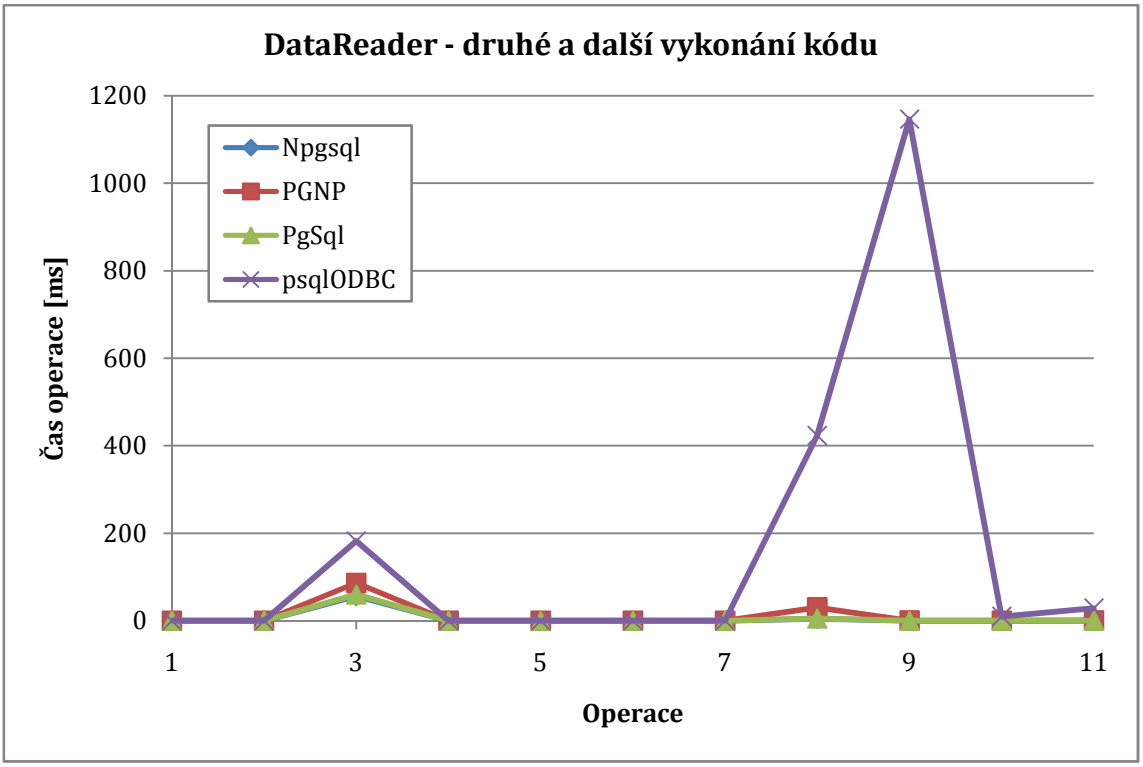
A Grafy časových náročností



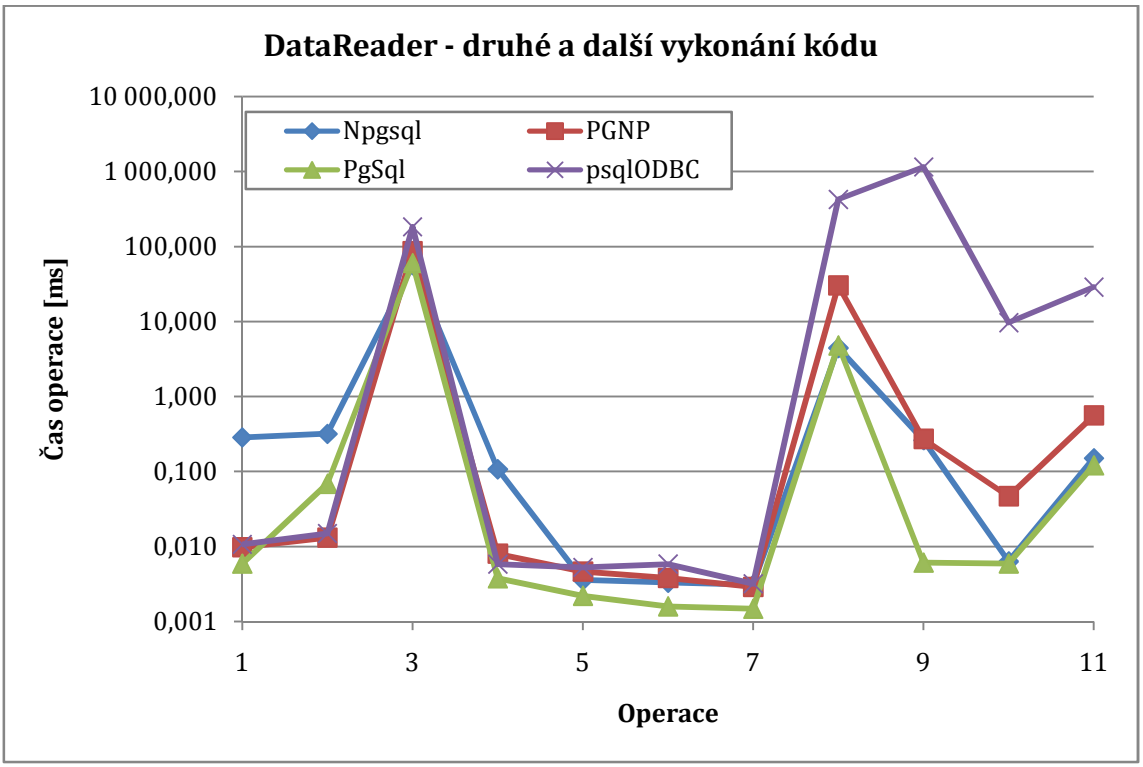
Obrázek A-1: Průměrná časová náročnost jednotlivých operací při prvním vykonání kódu s použitím objektu DataReader.



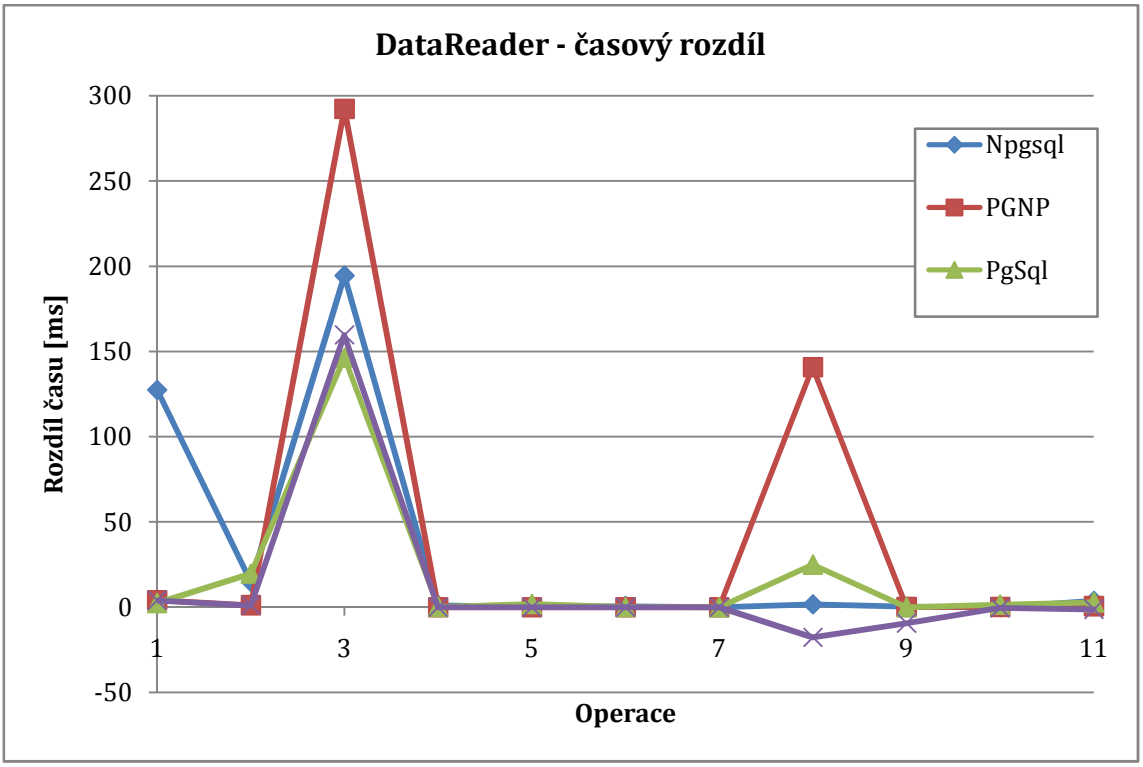
Obrázek A-2: Průměrná časová náročnost jednotlivých operací při prvním vykonání kódu s použitím objektu DataReader v logaritmickém měřítku.



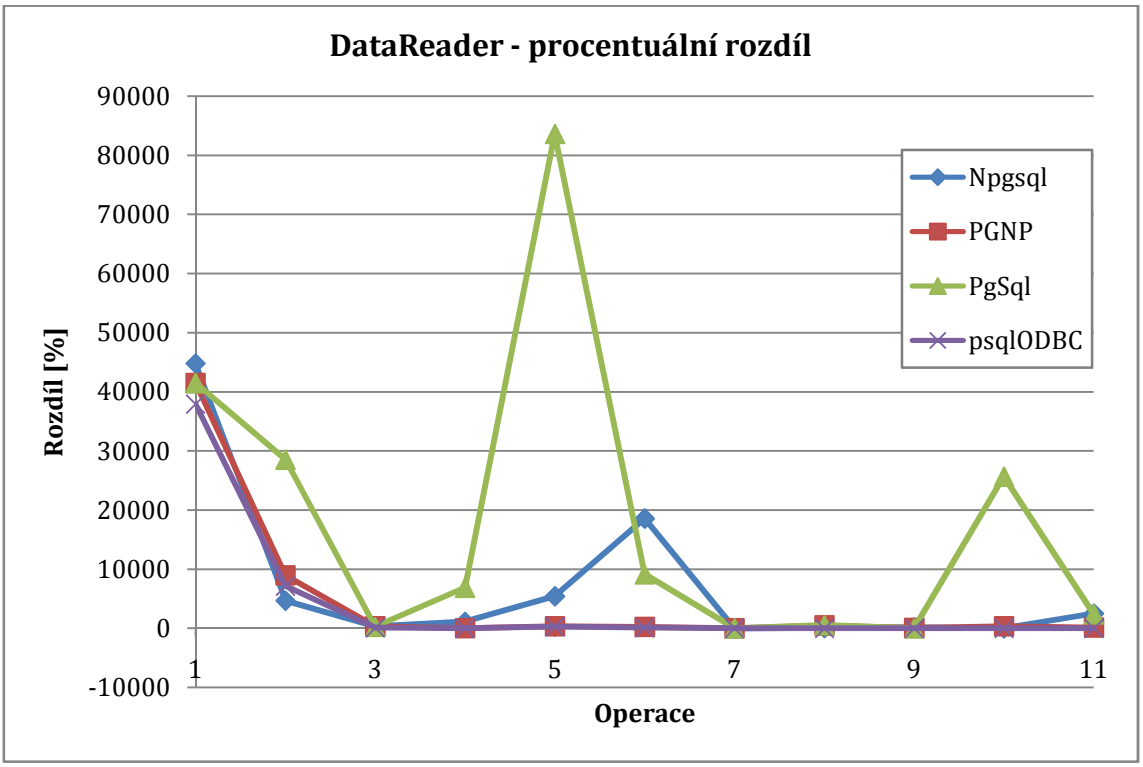
Obrázek A-3: Průměrná časová náročnost jednotlivých operací při druhém a dalším vykonání kódu s použitím objektu DataReader.



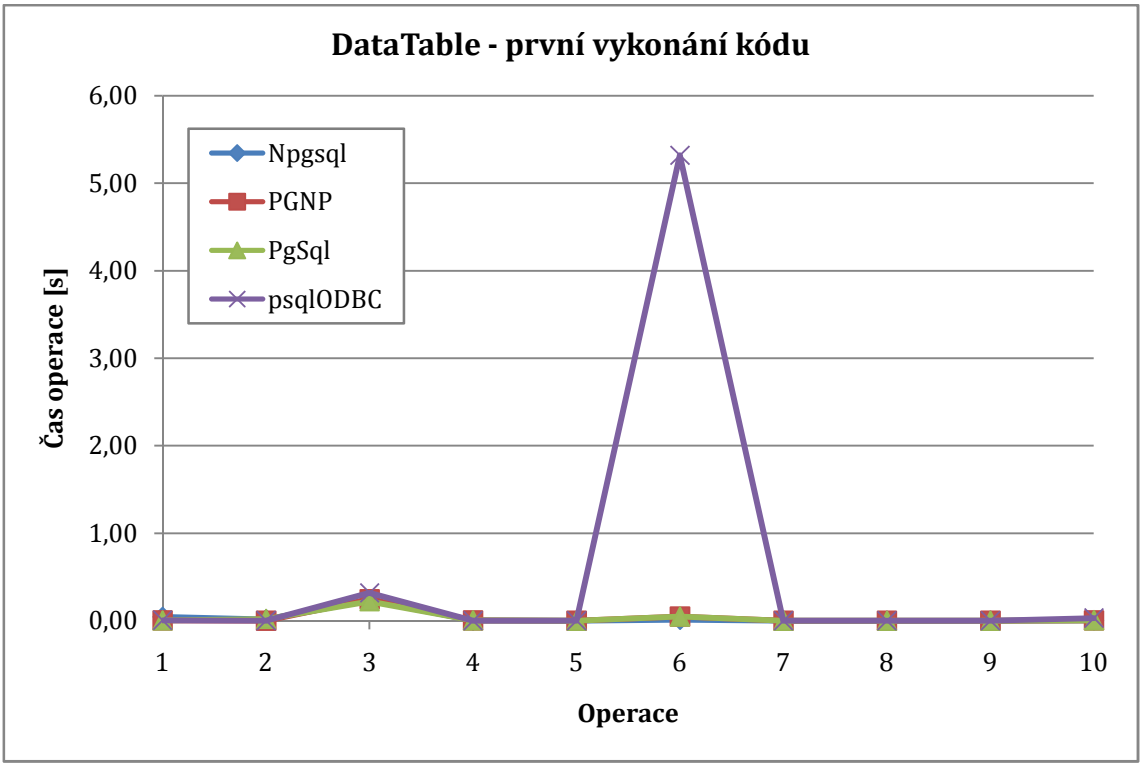
Obrázek A-4: Průměrná časová náročnost jednotlivých operací při druhém a dalším vykonání kódu s použitím objektu DataReader v logaritmickém měřítku.



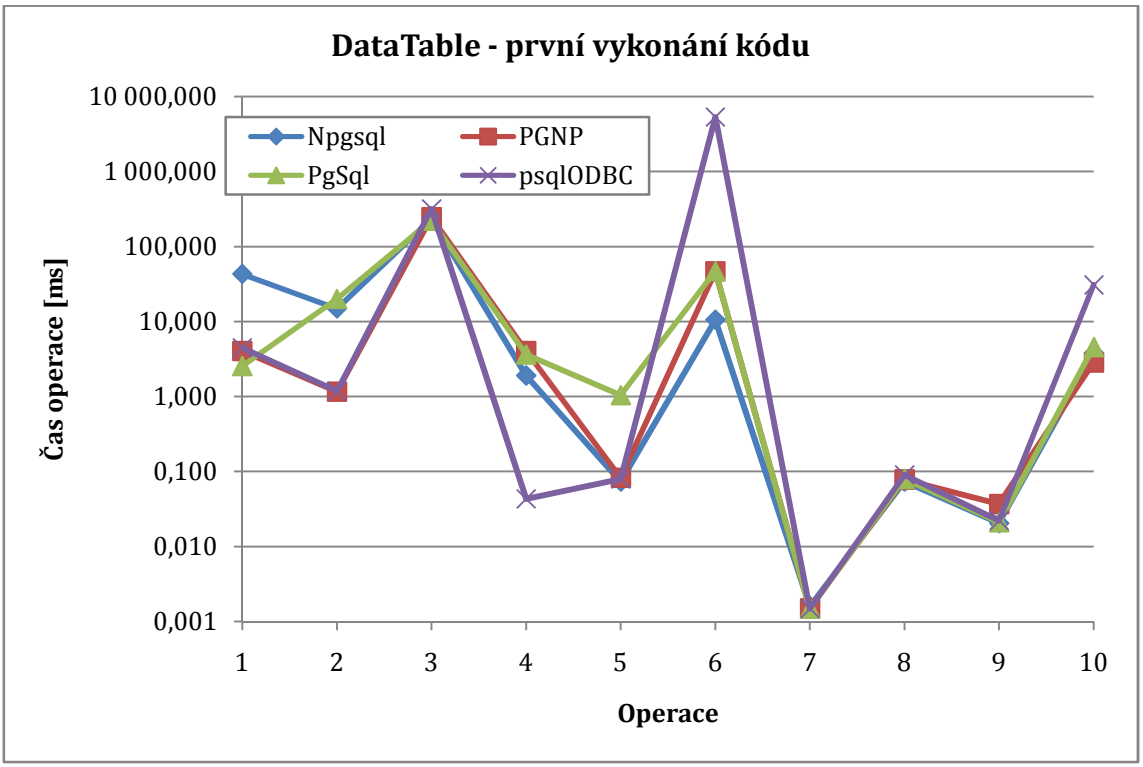
Obrázek A-5: Časový rozdíl mezi prvním a dalším vykonáním kódu s objektem DataReader.



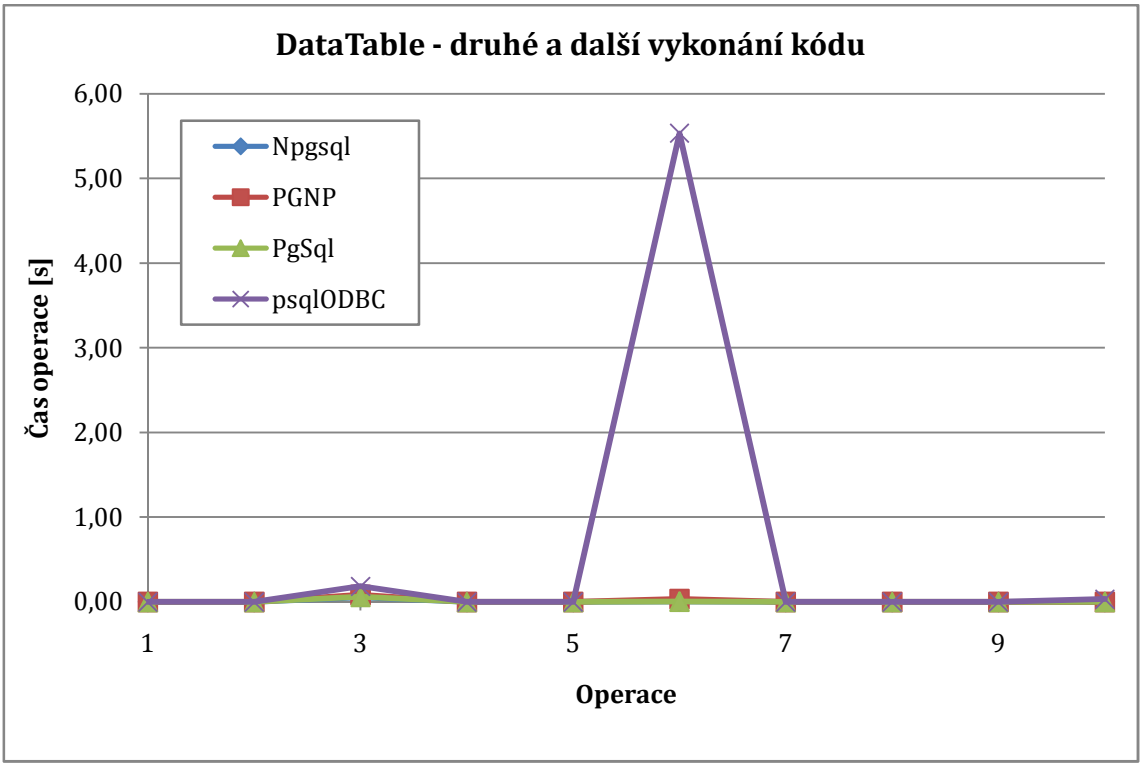
Obrázek A-6: Procentuální rozdíl časové náročnosti prvního vykonání kódu vůči průměrné hodnotě druhého a dalšího vykonání kódu s objektem DataReader.



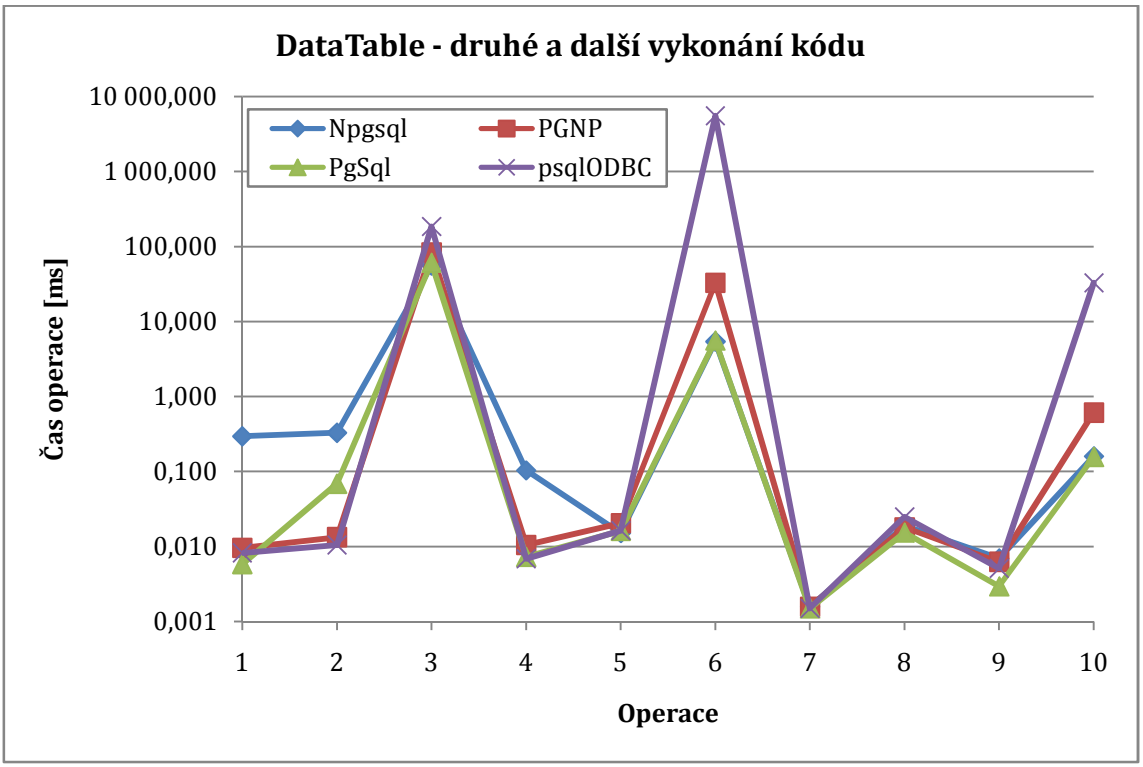
Obrázek A-7: Průměrná časová náročnost jednotlivých operací při prvním vykonání kódu s použitím objektu DataTable.



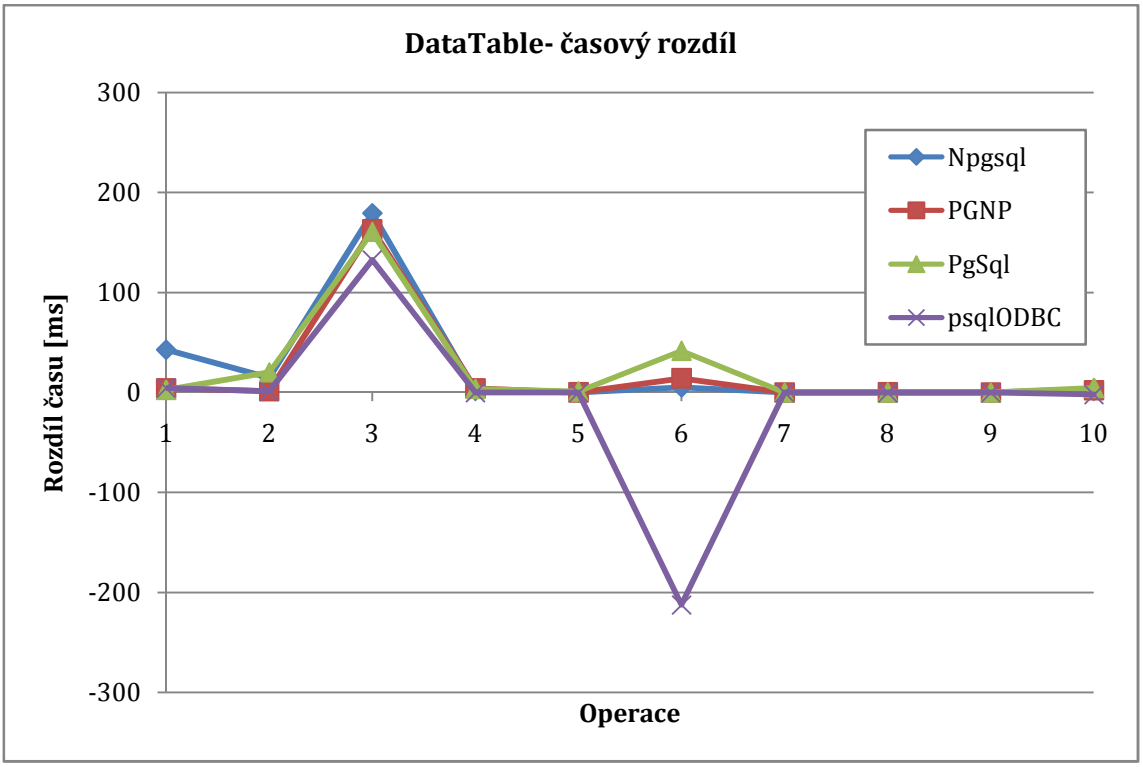
Obrázek A-8: Průměrná časová náročnost jednotlivých operací při prvním vykonání kódu s použitím objektu DataTable v logaritmickém měřítku.



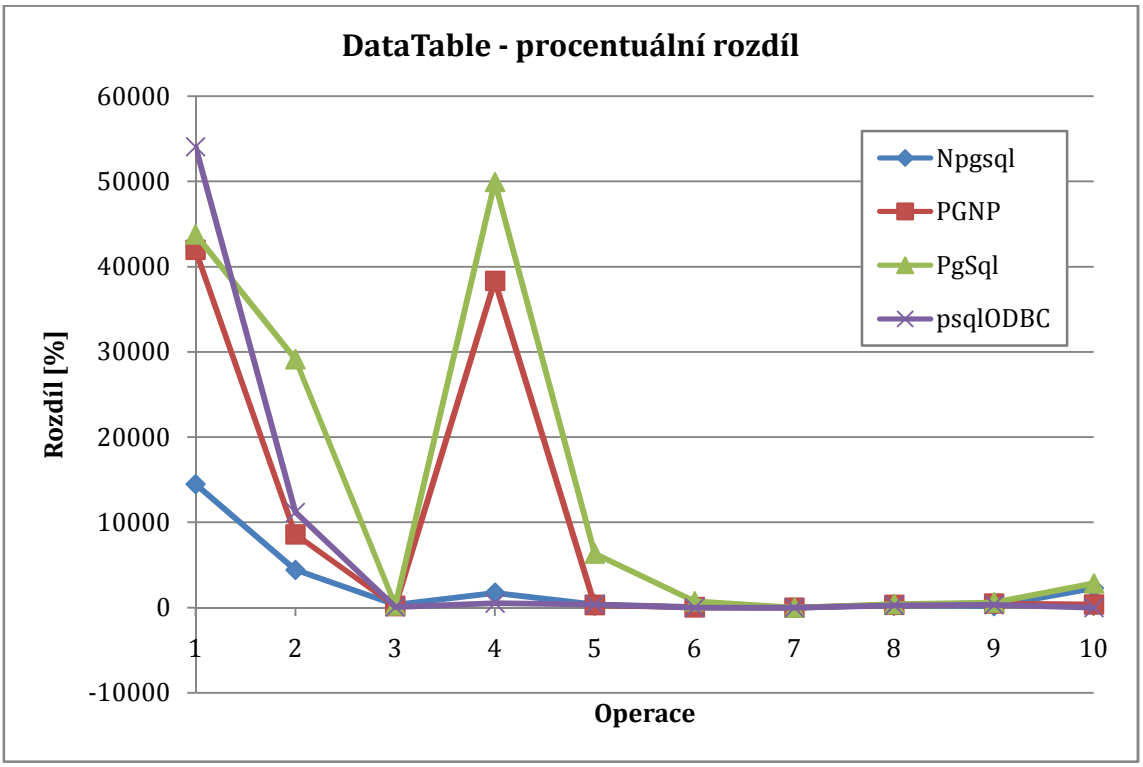
Obrázek A-9: Průměrná časová náročnost jednotlivých operací při druhém a dalším vykonání kódu s použitím objektu DataTable.



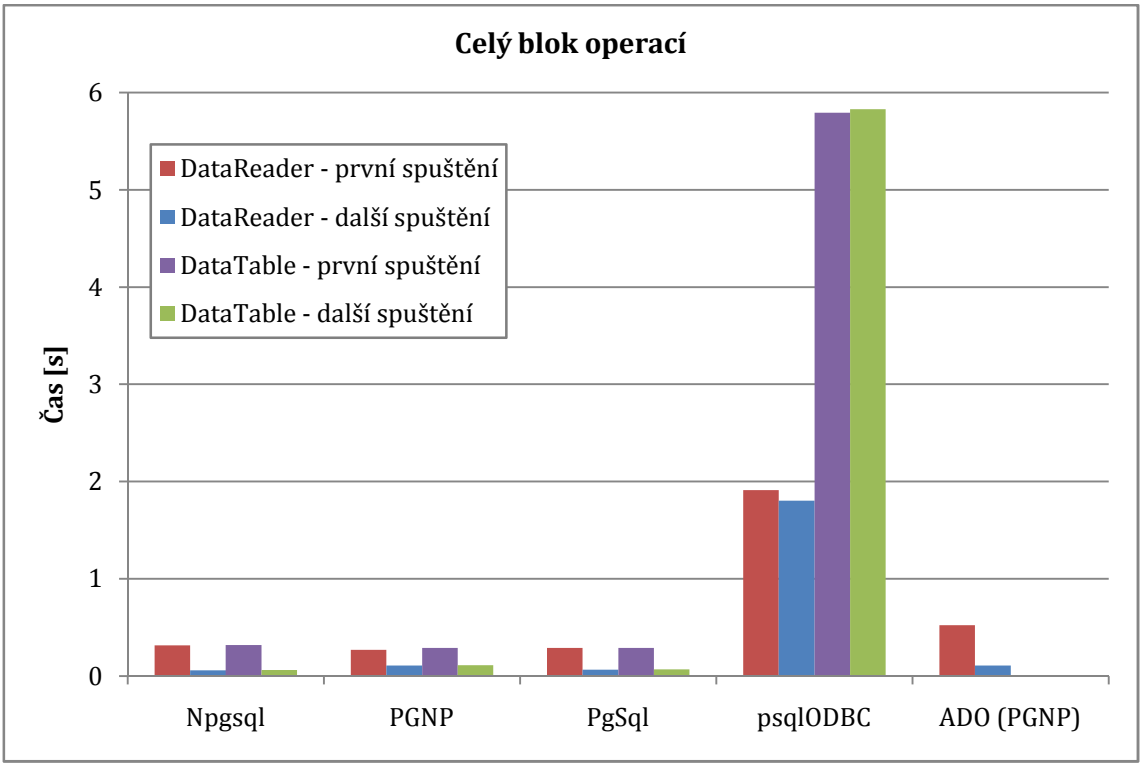
Obrázek A-10: Průměrná časová náročnost jednotlivých operací při druhém a dalším vykonání kódu s použitím objektu DataTable v logaritmickém měřítku.



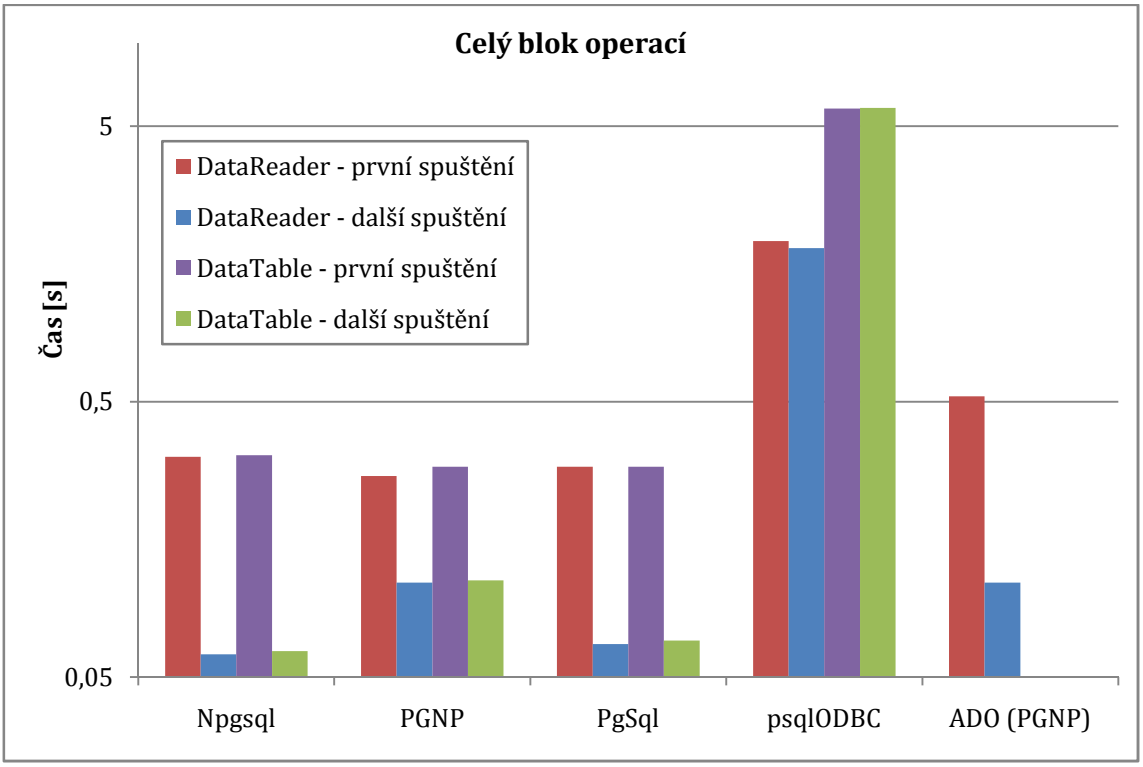
Obrázek A-11: Časový rozdíl mezi prvním a dalším vykonáním kódu s objektem DataTable.



Obrázek A-12: Procentuální rozdíl časové náročnosti prvního vykonání kódu vůči průměrné hodnotě druhého a dalšího vykonání kódu s objektem DataTable.



Obrázek A-13: Časová náročnost celého bloku operací.

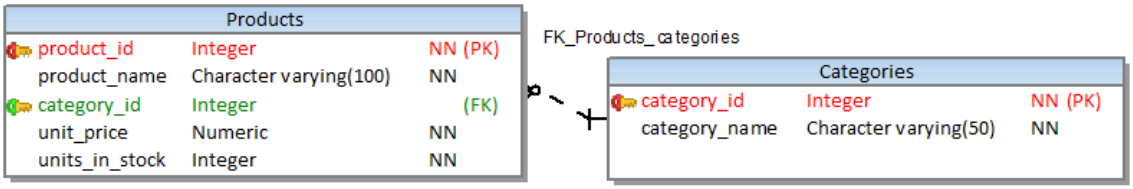


Obrázek A-14: Časová náročnost celého bloku operací v logaritmickém měřítku.

B Vzorové databázové tabulky

```
-- Tabulka kategorií
CREATE TABLE shop."Categories"
(
  category_id serial NOT NULL, -- ID
  category_name character varying(50) NOT NULL, -- Název kategorie
  CONSTRAINT "PK_Categories" PRIMARY KEY (category_id)
)
WITH (OIDS=FALSE);
ALTER TABLE shop."Categories" OWNER TO root;
COMMENT ON TABLE shop."Categories" IS 'Seznam kategorií pro produkty';
COMMENT ON COLUMN shop."Categories".category_id IS 'ID';
COMMENT ON COLUMN shop."Categories".category_name IS 'Název kategorie';

-- Tabulka produktů
CREATE TABLE shop."Products"
(
   product_id serial NOT NULL, -- ID
   product_name character varying(100) NOT NULL, -- Název produktu
   category_id integer, -- Kategorie produktu
   unit_price numeric NOT NULL DEFAULT 0, -- Jednotková cena za kus
   units_in_stock integer NOT NULL DEFAULT 0, -- Počet kusů skladem
   CONSTRAINT "PK_Products" PRIMARY KEY (product_id),
   CONSTRAINT "FK_Products_categories" FOREIGN KEY (category_id)
      REFERENCES shop."Categories" (category_id) MATCH SIMPLE
      ON UPDATE SET NULL ON DELETE SET NULL
)
WITH (OIDS=FALSE);
ALTER TABLE shop."Products" OWNER TO root;
COMMENT ON TABLE shop."Products" IS 'Tabulka produktů
Přiřazení do kategorie je nepovinné';
COMMENT ON COLUMN shop."Products".product_id IS 'ID';
COMMENT ON COLUMN shop."Products".product_name IS 'Název produktu';
COMMENT ON COLUMN shop."Products".category_id IS 'Kategorie produktu';
COMMENT ON COLUMN shop."Products".unit_price IS 'Jednotková cena za kus';
COMMENT ON COLUMN shop."Products".units_in_stock IS 'Počet kusů skladem';
```



Obrázek B-1 ER diagram vzorových databázových tabulek

C Vygenerovaný zdrojový kód modelu

```
namespace ShopModel
{
    using System;
    using Npgsql;
    using NpgsqlTypes;
    using MMST.NpgObjects;

    public partial class ShopNpgObjects : NpgObjects
    {
        public ShopNpgObjects(string connectionString) :
            base(connectionString, false)
        {}

        public ShopNpgObjects(NpgsqlConnection connection) :
            base(connection, false)
        {}

        public NpgTable<ShopCategory> ShopCategories
        {
            get
            {
                if (this._shopCategories == null)
                    this._shopCategories = base.GetTable<ShopCategory>();
                return this._shopCategories;
            }
        }
        private NpgTable<ShopCategory> _shopCategories;

        public NpgTable<ShopProduct> ShopProducts
        {
            get
            {
                if (this._shopProducts == null)
                    this._shopProducts = base.GetTable<ShopProduct>();
                return this._shopProducts;
            }
        }
        private NpgTable<ShopProduct> _shopProducts;
    }

    /// <summary>
    /// Schema: Schéma pro jednoduchý shop.
    /// Obsahuje tabulky Products a Categories pro seznam zboží a jeho kategorií.
    /// Table:  Seznam kategorií pro produkty
    /// </summary>
    [Table("Categories", "shop")]
    public partial class ShopCategory
    {
        /// <summary>
        /// ID
        /// <summary>
        [PrimaryKey(Column = "category_id", DbType = NpgsqlDbType.Integer,
            IsAutoIncrement = true)]
        public Int32 CategoryId { get; set; }

        /// <summary>
        /// Název kategorie
        /// <summary>
```

```
        [Property(Column = "category_name", DbType = NpgsqlDbType.Varchar,
            IsNullable = false)]
        public String CategoryName { get; set; }
    }

    /// <summary>
    /// Schema: Schéma pro jednoduchý shop.
    /// Obsahuje tabulky Products a Categories pro seznam zboží a jeho kategorií.
    /// Table:  Tabulka produktů
    /// Přiřazení do kategorie je nepovinné
    /// </summary>
    [Table("Products", "shop")]
    public partial class ShopProduct
    {
        /// <summary>
        /// ID
        /// <summary>
        [PrimaryKey(Column = "product_id", DbType = NpgsqlDbType.Integer,
            IsAutoIncrement = true)]
        public Int32 ProductId { get; set; }

        /// <summary>
        /// Název produktu
        /// <summary>
        [Property(Column = "product_name", DbType = NpgsqlDbType.Varchar,
            IsNullable = false)]
        public String ProductName { get; set; }

        /// <summary>
        /// Kategorie produktu
        /// <summary>
        [ForeignKey("Categories", "category_id", "shop", Column = "category_id",
            DbType = NpgsqlDbType.Integer, IsNullable = true)]
        public Int32? CategoryId { get; set; }

        /// <summary>
        /// Jednotková cena za kus
        /// <summary>
        [Property(Column = "unit_price", DbType = NpgsqlDbType.Numeric,
            IsNullable = false)]
        public Decimal UnitPrice { get; set; }

        /// <summary>
        /// Počet kusů skladem
        /// <summary>
        [Property(Column = "units_in_stock", DbType = NpgsqlDbType.Integer,
            IsNullable = false)]
        public Int32 UnitsInStock { get; set; }
    }
}
```

D XML dokument pro tvorbu modelu

```
<?xml version="1.0" encoding="utf-8"?>
<npgobjects>
  <database>mmst</database>
  <name>MmstNpgObjects</name>
  <useExtendedTypes>false</useExtendedTypes>
  <namespace>MmstModel</namespace>
  <date>2009-05-22 23:19:21Z</date>
  <objects>
    <object>
      <className>ShopCategory</className>
      <collectionName>ShopCategories</collectionName>
      <tableName>Categories</tableName>
      <tableSchema>shop</tableSchema>
      <commentTable>Seznam kategorií pro produkty</commentTable>
      <commentSchema>Schéma pro jednoduchý shop.
Obsahuje pouze dvě tabulky Products a Categories pro seznam zboží a jeho
      kategorií.</commentSchema>
      <privatePropertyName>_shopCategories</privatePropertyName>
      <isInsertable />
      <columns>
        <column>
          <name>category_id</name>
          <comment>ID</comment>
          <propertyName>CategoryId</propertyName>
          <type>Int32</type>
          <dbType>NpgsqlDbType.Integer</dbType>
          <autoIncrement />
          <primaryKey />
        </column>
        <column>
          <name>category_name</name>
          <comment>Název kategorie</comment>
          <propertyName>CategoryName</propertyName>
          <type>String</type>
          <dbType>NpgsqlDbType.Varchar</dbType>
        </column>
      </columns>
    </object>
    <object>
      <className>ShopProduct</className>
      <collectionName>ShopProducts</collectionName>
      <tableName>Products</tableName>
      <tableSchema>shop</tableSchema>
      <commentTable>Tabulka produktů
Přiřazení do kategorie je nepovinné</commentTable>
      <commentSchema>Schéma pro jednoduchý shop.
Obsahuje pouze dvě tabulky Products a Categories pro seznam zboží a jeho
      kategorií.</commentSchema>
      <privatePropertyName>_shopProducts</privatePropertyName>
      <isInsertable />
      <columns>
        <column>
          <name>product_id</name>
          <comment>ID</comment>
          <propertyName>ProductId</propertyName>
          <type>Int32</type>
          <dbType>NpgsqlDbType.Integer</dbType>
          <autoIncrement />
```

```
          <primaryKey />
        </column>
        <column>
          <name>product_name</name>
          <comment>Název produktu</comment>
          <propertyName>ProductName</propertyName>
          <type>String</type>
          <dbType>NpgsqlDbType.Varchar</dbType>
        </column>
        <column>
          <name>category_id</name>
          <comment>Kategorie produktu - nepovinná</comment>
          <propertyName>CategoryId</propertyName>
          <nullable />
          <type>Int32?</type>
          <dbType>NpgsqlDbType.Integer</dbType>
          <foreignKey>
            <schema>shop</schema>
            <table>Categories</table>
            <column>category_id</column>
          </foreignKey>
        </column>
        <column>
          <name>unit_price</name>
          <comment>Jednotková cena za kus bez DPH</comment>
          <propertyName>UnitPrice</propertyName>
          <type>Decimal</type>
          <dbType>NpgsqlDbType.Numeric</dbType>
        </column>
        <column>
          <name>units_in_stock</name>
          <comment>Počet kusů skladem</comment>
          <propertyName>UnitsInStock</propertyName>
          <type>Int32</type>
          <dbType>NpgsqlDbType.Integer</dbType>
        </column>
      </columns>
    </object>
  </objects>
</npgobjects>
```

E Tabulka datových typů

Tab. E-1: Vztah mezi databázovými typy, označením typů v Npgsql a datovými typy v CLR

PostgreSQL	Npgsql (enum)	CLR (not nullable)	CLR (nullable)
oidvector	NpgsqlDbType.Text	String	String
refcursor	NpgsqlDbType.Refcursor	String	String
char	NpgsqlDbType.Char	String	String
bpchar	NpgsqlDbType.Text	String	String
varchar	NpgsqlDbType.Varchar	String	String
text	NpgsqlDbType.Text	String	String
name	NpgsqlDbType.Text	String	String
bytea	NpgsqlDbType.Bytea	Byte[]	Byte[]
bit	NpgsqlDbType.Bit	BitString	BitString?
bool	NpgsqlDbType.Boolean	Boolean	Boolean?
int2	NpgsqlDbType.Smallint	Int16	Int16?
int4	NpgsqlDbType.Integer	Int32	Int32?
int8	NpgsqlDbType.Bigint	Int64	Int64?
oid	NpgsqlDbType.Bigint	Int64	Int64?
float4	NpgsqlDbType.Real	Single	Single?
float8	NpgsqlDbType.Double	Double	Double?
numeric	NpgsqlDbType.Numeric	Decimal	Decimal?
inet	NpgsqlDbType.Inet	NpgsqlInet	NpgsqlInet?
money	NpgsqlDbType.Money	Decimal	Decimal?
point	NpgsqlDbType.Point	NpgsqlPoint	NpgsqlPoint?
lseg	NpgsqlDbType.LSeg	NpgsqlLSeg	NpgsqlLSeg?
path	NpgsqlDbType.Path	NpgsqlPath	NpgsqlPath?
box	NpgsqlDbType.Box	NpgsqlBox	NpgsqlBox?
circle	NpgsqlDbType.Circle	NpgsqlCircle	NpgsqlCircle?
polygon	NpgsqlDbType.Polygon	NpgsqlPolygon	NpgsqlPolygon?
uuid	NpgsqlDbType.Uuid	Guid	Guid?
xml	NpgsqlDbType.Xml	String	String
interval	NpgsqlDbType.Interval	NpgsqlInterval	NpgsqlInterval?
date	NpgsqlDbType.Date	DateTime	DateTime?
time	NpgsqlDbType.Time	DateTime	DateTime?
timetz	NpgsqlDbType.TimeTZ	DateTime	DateTime?
timestamp	NpgsqlDbType.Timestamp	DateTime	DateTime?
timestamptz	NpgsqlDbType.TimestampTZ	DateTime	DateTime?

F Příklady použití vytvořených knihoven

Součástí *solution MMST* na přiloženém DVD jsou tři projekty s příklady použití vytvořených knihoven `NpgObjects.dll` a `PagedDataGridView.dll`. Zde jsou shrnuty hlavní body těchto příkladů.

Projekt ExampleWebShop

Projekt ExampleWebShop je ukázkou použití *NpgObjects* v prostředí webové aplikace postavené na **ASP.NET MVC.** Zvolený příklad je ukázkou primitivní administrace internetového obchodu.

Úvodní stránka zobrazuje data získaná z pohledu `shop."Products with Category and VAT"`, který zobrazuje data tabulky `shop."Products"` společně s názvem kategorie získaným z tabulky `shop."Categories"` a dopočítanou jednotkovou cenou s DPH.

Název	Kategorie	Cena s (bez) DPH	Kusů	
L. Lacko: SQL - Kapesní přehled	Kniha	**89,00 Kč** (74,79 Kč)	3	Upravit | Odstranit
Twin Peaks	Filmové DVD	**99,00 Kč** (83,19 Kč)	13	Upravit | Odstranit
Mulholland Drive	Filmové DVD	**99,00 Kč** (83,19 Kč)	12	Upravit | Odstranit
Příběh Alvina Straighta	Filmové DVD	**99,00 Kč** (83,19 Kč)	14	Upravit | Odstranit
Gothart - Optimi de...	Hudební CD	**149,00 Kč** (125,21 Kč)	3	Upravit | Odstranit
Walk the Line	Filmové DVD	**185,00 Kč** (155,46 Kč)	4	Upravit | Odstranit
J. R. R. Tolkien: Silmarillion	Kniha	**190,00 Kč** (159,66 Kč)	2	Upravit | Odstranit
J. R. R. Tolkien: Hobit	Kniha	**190,00 Kč** (159,66 Kč)	15	Upravit | Odstranit
Moravanka - Zlatá Moravanka	Hudební CD	**199,00 Kč** (167,23 Kč)	3	Upravit | Odstranit
Divokej Bill - Lucerna Live	Hudební CD	**269,00 Kč** (226,05 Kč)	11	Upravit | Odstranit

Obr. F-1: Úvodní stránka webové aplikace používající NpgObjects

Nad tabulkou `shop."Products"` lze provádět všechny základní operace, jako je přidání, editace a odstranění záznamu. Díky NpgObjects a MVC frameworku je to záležitost několika málo řádků, jak ukazuje následující kód pro přidání nového záznamu do databáze.

```
// nový objekt
var product = new ShopProduct();
// seznam vlastností, které je povoleno insertovat
var whiteList = new[] { "ProductName", "UnitPrice",
                        "UnitsInStock", "CategoryId" };
// snaplnění objektu daty z formuláře po metodě POST
UpdateModel(product, whiteList);
// vložení nového objektu do tabulky a uložení (_shop je datový kontext)
_shop.ShopProducts.Insert(product);
_shop.ShopProducts.SaveChanges();
```

Projekt ExampleIPagedDataSourceConsole

Tento projekt je ukázkou použití „kolekce“ NpgPagedCollection<T> implementující rozhraní IPagedDataSource v prostředí konzolové aplikace.

```
// z tabulky tests vybere 100 záznamů a bude je stránkovat po 20
var data = testNpgObjects.Tests.Where("id < 100").Select(20);
do
{
  // vypíše všechny řádky stránky
    foreach (var x in data)
    {
         Console.WriteLine(x.Id);
    }
    // po každé stránce čeká na uživatelský vstup
    Console.ReadLine();
// stránkuje, dokud je nějaká další stránka
} while (data.NextPage());
```

Uvedený kód je proti skutečnému příkladu pro přehlednost zjednodušen, aby lépe vynikla základní myšlenka postupného vypisování dat do konzole pomocí stránkovací kolekce NpgPagedCollection projektu NpgObjects.

Projekt ExamplesPagedDataGridView

Tento projekt obsahuje čtyři formuláře pro představení spolupráce komponenty PagedDataGridView s různými zdroji dat a pro představení možností jejího nastavení.

Formulář Pdgv1PagedSource

Představuje základní použití PagedDataGridView pro datový zdroj implementující rozhraní IPagedDataSource.

Pomocí změny filtru na schodu počátečního písmena se mění množina dat, nad kterou se stránkování provádí:

```
pagedDataGrigView.DataSource = testNpgObjects.Tests
   .Where(" a LIKE @filter", new NpgsqlParameter("filter", letter + "%" ))
   .Select(_pageSize);
```

Formulář Pdgv2PagedSource

Ukazuje pokročilejší možnosti práce s PagedDataGridView, využívá jeho události pro editaci záznamů. Následující (opět zjednodušený) příklad za všechny představuje obsluhu události mazání řádku z tabulky.

```
private void pagedDataGrigView_DeletingRow(object sender,
                                           DataGridViewRowCancelEventArgs e)
{
    // získá objekt ke smazání
    var currentItem = (Tests)pagedDataGrigView.CurrentRowDataBoundItem;
    // přidá objekt do seznamu mazaných a smaže uložením změn
    testNpgObjects.Tests.Delete(currentItem);
    testNpgObjects.Tests.SaveChanges();
}
```

Formulář Pdgv3NpgCollection

Demonstruje především možnosti nastavení různých vlastností komponenty PagedDataGridView, dále definici vlastních sloupců v gridu a konečně spolupráci s nestránkujícím zdrojem dat v podobě NpgCollection<T>.

Formulář Pdgv4AnonymousClasses

Ukazuje jedinou věc, a to, že si PagedDataGridView poradí v zobrazení i daty, které vůbec nepochází z projektu NpgObjects – zobrazuje seznam do paměti počítače vygenerovaných anonymních tříd.

G Obsah přiloženého DVD

Na přiloženém DVD se nachází soubor README.txt s informacemi o DVD a následující adresáře:

- **bin**

  Obsahuje binární verze všech příkladů a knihoven.

- **doc**

  Obsahuje tento text ve formátu PDF a návod k instalaci testovací databáze.

- **external**

  Obsahuje veškeré knihovny a další soubory stažené z internetu, které byly při realizaci této práce použity.

- **install**

  Obsahuje soubory pro instalaci testovací databáze.

- **src**

  Obsahuje solution MMST, ve kterém jsou obsaženy všechny projekty vytvořené pro knihovnu NpgObjects, generátor a komponentu PagedDataGridView a dále obsahuje projekty s příklady využívajícími vytvořené knihovny.